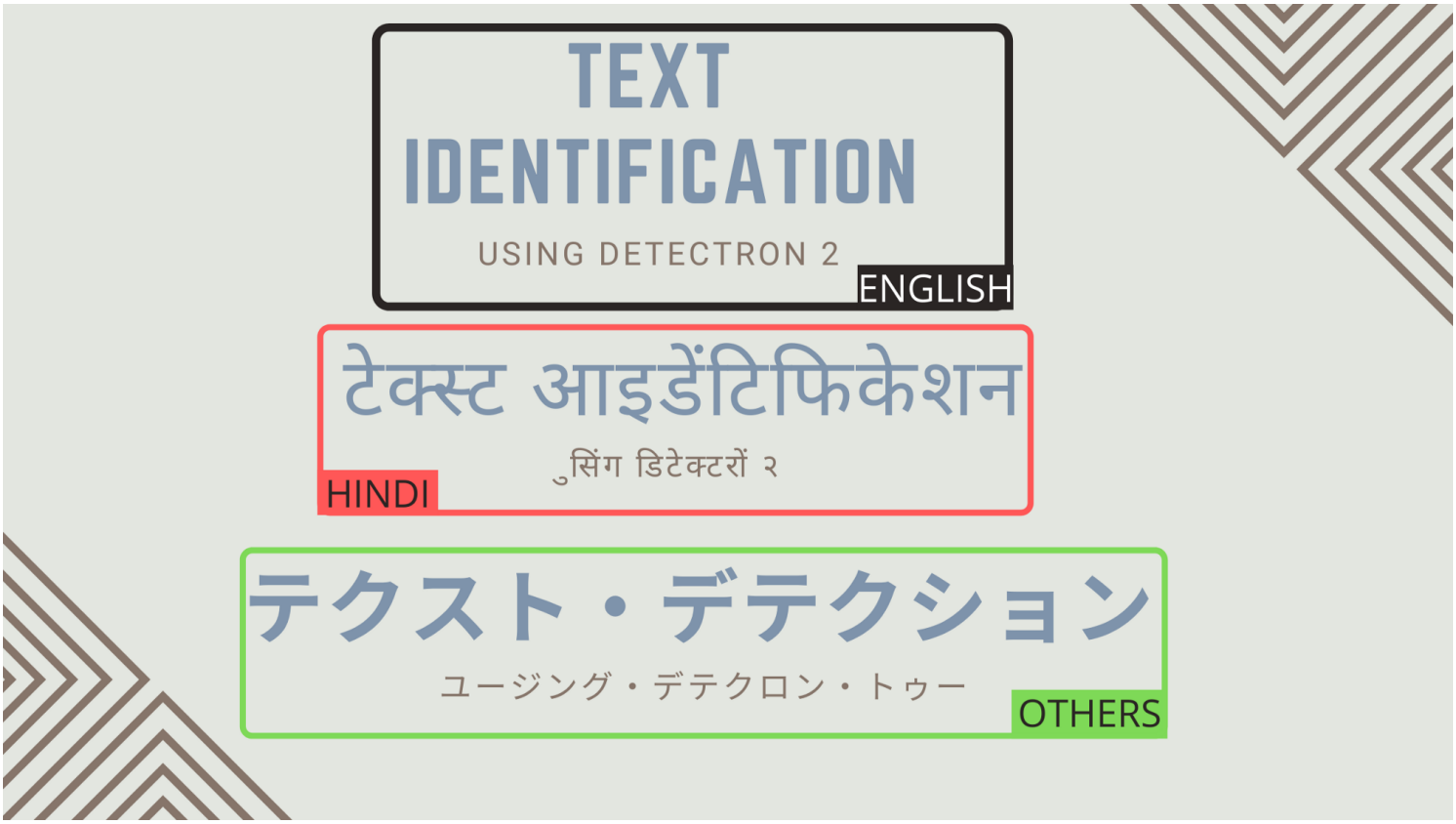
Object Detection in 6 steps using Detectron2

Let's look at how to use FAIR's (Facebook AI Research) Detectron 2 for Instance Detection on a custom dataset which involves text identification.

Aakarsh Yelisetty Aug 2, 2020 · 7 min read ★



Designed using Canva

Have you ever tried training an object detection model using a custom dataset of your own choice from scratch?

If yes, you'd know how tedious the process would be. We need to start with building a model using a Feature Pyramid Network combined with a Region Proposal Network if we opt for region proposal based methods such as Faster R-CNN or we can also use one-shot detector algorithms like SSD and YOLO.

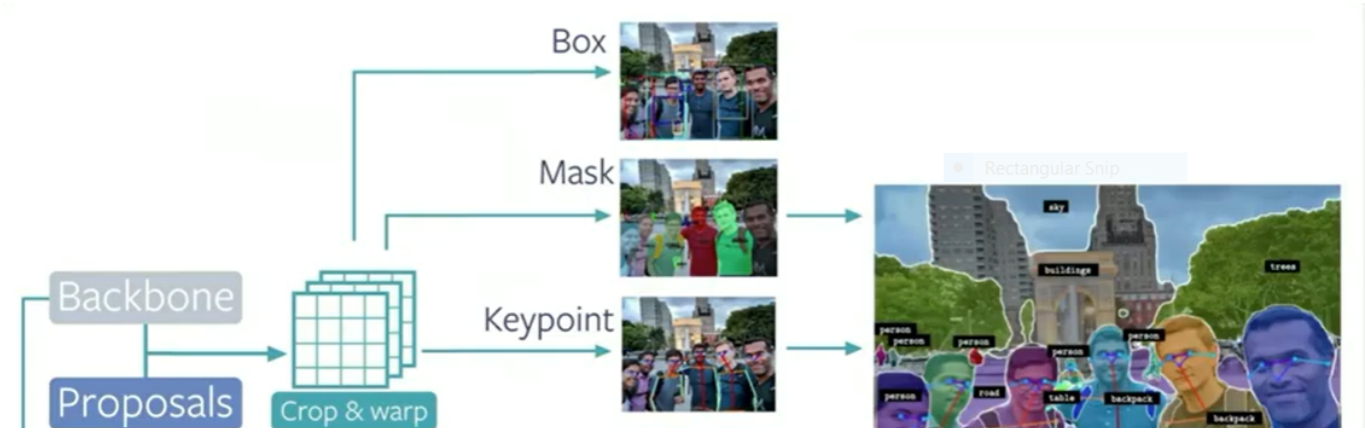
Either of them is a bit complicated to work with if we want to implement it from scratch. We need a framework where we can use state-of-the-art models such as Fast, Faster, and Mask R-CNNs with ease. Nevertheless, it is important to try building a model at least once from scratch to understand the math behind it.

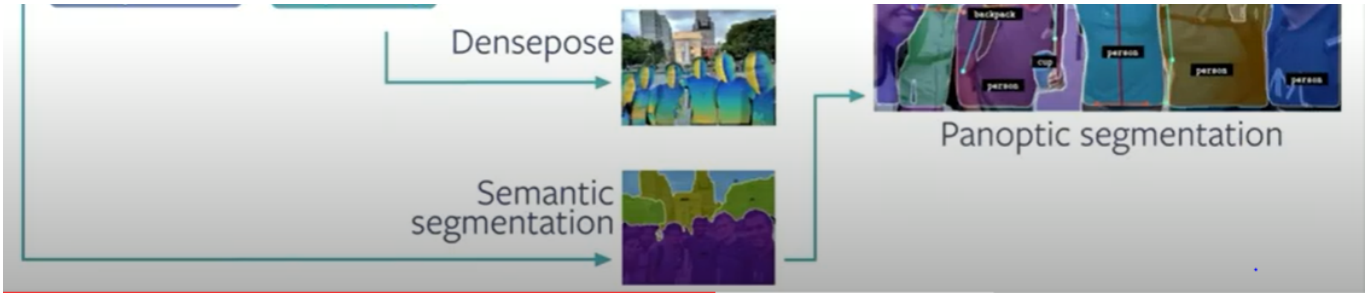
Detectron 2 comes to the rescue if we want to train an object detection model in a snap with a custom dataset. All the models present in the model zoo of the Detectron 2 library are pre-trained on COCO Dataset. We just need to fine-tune our custom dataset on the pre-trained model.

Detectron 2 is a complete rewrite of the first Detectron which was released in the year 2018. The predecessor was written on Caffe2, a deep learning framework that is also backed by Facebook. Both the Caffe2 and Detectron are now deprecated. Caffe2 is now a part of PyTorch and the successor, Detectron 2 is completely written on PyTorch.

> *Detectron2 is meant to advance machine learning by offering speedy training and addressing the issues companies face when making the step from research to production.*

These are the various types of Object Detection models that the Detectron 2 offers.





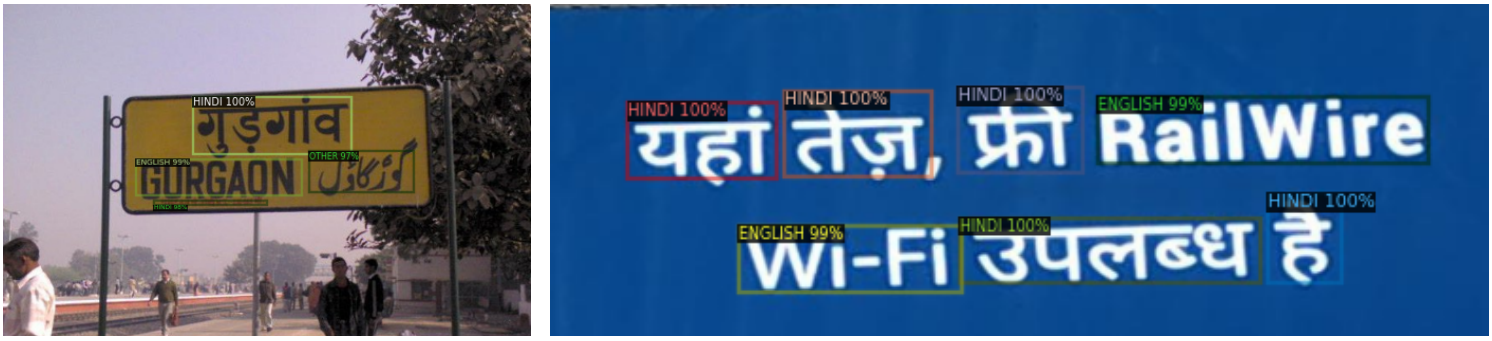
https://research.fb.com/wp-content/uploads/2019/12/4.-detectron2.pdf

Let's dive into **Instance Detection** directly.

Instance Detection refers to the classification and localization of an object with a bounding box around it. In this article, **We are going to deal with identifying the language of text from images using the Faster RCNN model from the Detectron 2's model zoo.**

Note that we are going to limit our languages by 2.

We identify **Hindi** and **English** Text and we include a class labeled **Others** for other languages.



Final Results from Colab

We'll implement a model where the output would be in this way.

Let's get started!

Using Detectron 2, Object Detection can be performed on any custom dataset using seven steps. All the steps are readily available in this Google Colab Notebook and you can run it straight away!

Using Google Colab for this would be an easy task as we can use a GPU for faster training.

Step 1: Installing Detectron 2

Kickstart with installing a few dependencies such as Torch Vision and COCO API and check whether **CUDA** is available. CUDA helps in keeping track of the currently selected **GPU**. And then install Detectron2.

```
# install dependencies:
!pip install -U torch==1.5 torchvision==0.6 -f
https://download.pytorch.org/whl/cu101/torch_stable.html
!pip install cython pyyaml==5.1
!pip install -U
'git+https://github.com/cocodataset/cocoapi.git#subdirectory=PythonA
PI'

import torch, torchvision
print(torch.__version__, torch.cuda.is_available())
!gcc --version

# install detectron2:
!pip install detectron2==0.1.3 -f
https://dl.fbaipublicfiles.com/detectron2/wheels/cu101/torch1.5/inde
x.html
```

Step 2: Prepare and Register the Dataset

Import a few necessary packages.

```
# You may need to restart your runtime prior to this, to let your installation take effe
import detectron2
from detectron2.utils.logger import setup_logger
setup_logger()

# import some common libraries
import numpy as np
import cv2
```

```
import cv2
import random
from google.colab.patches import cv2_imshow

# import some common detectron2 utilities
from detectron2 import model_zoo
from detectron2.engine import DefaultPredictor
from detectron2.config import get_cfg
from detectron2.utils.visualizer import Visualizer
from detectron2.data import MetadataCatalog
```

imports.py hosted with ❤️ by GitHub view raw

Datasets that have builtin support in detectron2 are listed in builtin datasets. If you want to use a custom dataset while also reusing detectron2's data loaders, you will need to **Register** your dataset (i.e., tell detectron2 how to obtain your dataset).

We use our Text Detection Dataset which has three classes:

1. English
2. Hindi
3. Others

We'll train a text detection model from an existing model pre-trained on the COCO dataset, available in detectron2's model zoo.

If you're interested to know the conversion from the raw format of the dataset to the format that Detectron 2 accepts, checkout this Colab Notebook.

There are certain formats in how data can be fed to a model such as a YOLO format, PASCAL VOC format, COCO format, etc. Detectron2 accepts the COCO Format of the dataset. COCO format of the dataset consists of a JSON file which includes all the details of an image such as size, annotations (i.e., bounding box coordinates), labels corresponding to it's bounding box, etc. For example,


```
{
  "image_id": 0,
  "file_name": "401.jpeg",
  "height": 500,
```

```
  "width": 715,
  "annotations": [
    {
      "bbox": [
        67,
        200,
        187,
        59
      ],
      "bbox_mode": 1,
      "category_id": "0"
    },
    {
      "bbox": [
        279,
        170,
        417,
        38
      ],
      "bbox_mode": 1,
      "category_id": "0"
    }
  ]
}
```

Detectron 2's format of Dataset

This is how the JSON looks like for one image. There are different types of formats for the bounding box representation. It must be a member of structures.BoxMode for Detectron2. There are 5 such formats. But currently, it supports **BoxMode.XYXY_ABS, BoxMode.XYWH_ABS**. We use the second format. The (X, Y) represents one coordinate of the bounding box, and W, H represents Width and Height of that box. The category_id refers to the class the box belongs to.

Then, we need to register our dataset.

```
import json
from detectron2.structures import BoxMode
def get_board_dicts(imgdir):
    json_file = imgdir+"/dataset.json" #Fetch the json file
    with open(json_file) as f:
        dataset_dicts = json.load(f)
    for i in dataset_dicts:
```

```
        filename = i["file_name"]
        i["file_name"] = imgdir+"/"+filename
        for j in i["annotations"]:
            j["bbox_mode"] = BoxMode.XYWH_ABS #Setting the required Box Mode
            j["category_id"] = int(j["category_id"])
    return dataset_dicts
from detectron2.data import DatasetCatalog, MetadataCatalog
#Registering the Dataset
for d in ["train", "val"]:
    DatasetCatalog.register("boardetect_" + d, lambda d=d: get_board_dicts("Text_Detecti
    MetadataCatalog.get("boardetect_" + d).set(thing_classes=["HINDI","ENGLISH","OTHER"]
board_metadata = MetadataCatalog.get("boardetect_train")
```

get_board_dicts.py hosted with ❤️ by GitHub view raw

To verify the data loading is correct, let's visualize the annotations of randomly selected samples in the training set.

Step 3: Visualize the Training Set

We'll randomly pick 3 pictures from the train folder of our dataset and see how the bounding boxes look like.

```
#Visualizing the Train Dataset
dataset_dicts = get_board_dicts("Text_Detection_Dataset_COCO_Format/train")
#Randomly choosing 3 images from the Set
for d in random.sample(dataset_dicts, 3):
    img = cv2.imread(d["file_name"])
    visualizer = Visualizer(img[:, :, ::-1], metadata=board_metadata)
    vis = visualizer.draw_dataset_dict(d)
    cv2_imshow(vis.get_image()[:, :, ::-1])
```

visualize.py hosted with ❤️ by GitHub view raw

The output looks in this way,

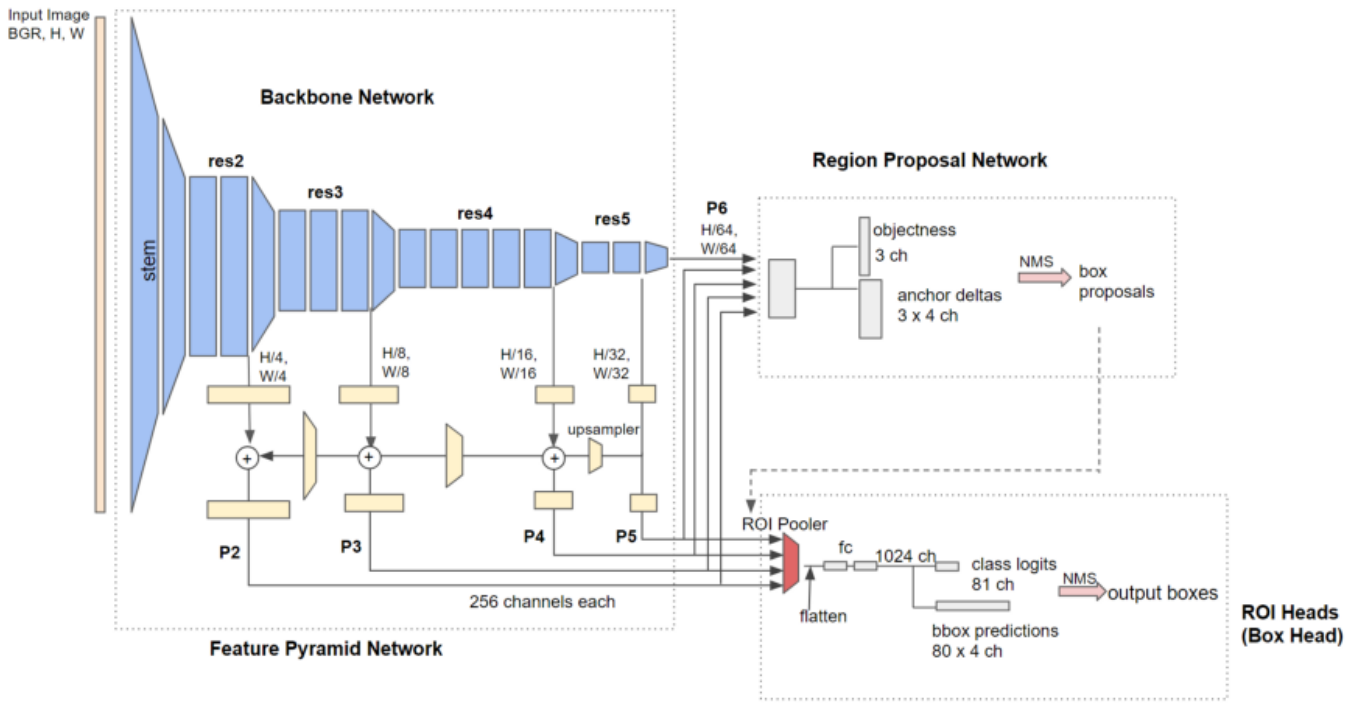
Results from Colab

Step 4: Training the Model

The big step. This is the step where we give configurations and set the model ready to get trained. Technically, we just fine-tune our model on the dataset as the model is

already pre-trained on COCO Dataset.

There are a ton of models available for object detection in the Detectron2's Model Zoo. Here, we use the **faster_rcnn_R_50_FPN_3x** model which looks in this way on a high level.



Source

There'd be a Backbone Network (Resnet in this case) which is used to extract features from the image followed by a Region Proposal Network for proposing region proposals and a Box Head for tightening the bounding box.

You can read more about how Faster R-CNN works in my previous article.

Let's set the configuration for training.

```
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-Detection/faster_rcnn_R_50_FPN_3x.ya
#Passing the Train and Validation sets
cfg.DATASETS.TRAIN = ("boardetect_train",)
cfg.DATASETS.TEST = ("boardetect_val",)
```

```
# Number of data loading threads
cfg.DATALOADER.NUM_WORKERS = 4
cfg.MODEL.WEIGHTS = model_zoo.get_checkpoint_url("COCO-Detection/faster_rcnn_R_50_FPN_3x
# Number of images per batch across all machines.
cfg.SOLVER.IMS_PER_BATCH = 4
cfg.SOLVER.BASE_LR = 0.0125  # pick a good LearningRate
cfg.SOLVER.MAX_ITER = 1500  #No. of iterations
cfg.MODEL.ROI_HEADS.BATCH_SIZE_PER_IMAGE = 256
cfg.MODEL.ROI_HEADS.NUM_CLASSES = 3 # No. of classes = [HINDI, ENGLISH, OTHER]
cfg.TEST.EVAL_PERIOD = 500 # No. of iterations after which the Validation Set is evaluat
os.makedirs(cfg.OUTPUT_DIR, exist_ok=True)
trainer = CocoTrainer(cfg)
trainer.resume_or_load(resume=False)
trainer.train()
```

set_config.py hosted with ❤️ by GitHub view raw

I wouldn't say this is the best configuration. Surely, the accuracy may get improved for other configs as well. After all, it depends on choosing the right hyperparameters.

```
cfg.MODEL.ROI_HEADS.NUM_CLASSES = 3 # No. of classes = [HINDI, ENGLISH, OTHER]
cfg.TEST.EVAL_PERIOD = 500
os.makedirs(cfg.OUTPUT_DIR, exist_ok=True)
trainer = CocoTrainer(cfg)
trainer.resume_or_load(resume=False)
trainer.train()
```

```
[08/01 12:28:15 d2.utils.events]:  eta: 0:25:53  iter: 39  total_loss: 1.522  loss_cls: 0.602  loss_box_reg: 0.758  loss_rpn_cls: 0.055  loss_rpn_loc: 0.056  time: 1.0614  data_time
[08/01 12:28:36 d2.utils.events]:  eta: 0:25:06  iter: 59  total_loss: 1.295  loss_cls: 0.395  loss_box_reg: 0.712  loss_rpn_cls: 0.044  loss_rpn_loc: 0.062  time: 1.0534  data_time
[08/01 12:28:57 d2.utils.events]:  eta: 0:24:48  iter: 79  total_loss: 1.110  loss_cls: 0.366  loss_box_reg: 0.632  loss_rpn_cls: 0.026  loss_rpn_loc: 0.051  time: 1.0476  data_time
[08/01 12:29:18 d2.utils.events]:  eta: 0:24:31  iter: 99  total_loss: 1.007  loss_cls: 0.351  loss_box_reg: 0.478  loss_rpn_cls: 0.041  loss_rpn_loc: 0.072  time: 1.0517  data_time
[08/01 12:29:39 d2.utils.events]:  eta: 0:24:14  iter: 119  total_loss: 0.804  loss_cls: 0.251  loss_box_reg: 0.400  loss_rpn_cls: 0.029  loss_rpn_loc: 0.059  time: 1.0510  data_tim
[08/01 12:30:00 d2.utils.events]:  eta: 0:23:48  iter: 139  total_loss: 0.738  loss_cls: 0.230  loss_box_reg: 0.397  loss_rpn_cls: 0.030  loss_rpn_loc: 0.053  time: 1.0483  data_tim
[08/01 12:30:21 d2.utils.events]:  eta: 0:23:28  iter: 159  total_loss: 0.694  loss_cls: 0.229  loss_box_reg: 0.383  loss_rpn_cls: 0.027  loss_rpn_loc: 0.050  time: 1.0494  data_tim
[08/01 12:30:41 d2.utils.events]:  eta: 0:23:03  iter: 179  total_loss: 0.688  loss_cls: 0.224  loss_box_reg: 0.351  loss_rpn_cls: 0.024  loss_rpn_loc: 0.039  time: 1.0466  data_tim
[08/01 12:31:03 d2.utils.events]:  eta: 0:22:45  iter: 199  total_loss: 0.758  loss_cls: 0.247  loss_box_reg: 0.388  loss_rpn_cls: 0.022  loss_rpn_loc: 0.046  time: 1.0475  data_tim
[08/01 12:31:23 d2.utils.events]:  eta: 0:22:21  iter: 219  total_loss: 0.631  loss_cls: 0.221  loss_box_reg: 0.350  loss_rpn_cls: 0.015  loss_rpn_loc: 0.043  time: 1.0467  data_tim
[08/01 12:31:44 d2.utils.events]:  eta: 0:22:00  iter: 239  total_loss: 0.654  loss_cls: 0.200  loss_box_reg: 0.346  loss_rpn_cls: 0.017  loss_rpn_loc: 0.044  time: 1.0454  data_tim
[08/01 12:32:05 d2.utils.events]:  eta: 0:21:38  iter: 259  total_loss: 0.581  loss_cls: 0.194  loss_box_reg: 0.340  loss_rpn_cls: 0.017  loss_rpn_loc: 0.044  time: 1.0444  data_tim
[08/01 12:32:25 d2.utils.events]:  eta: 0:21:20  iter: 279  total_loss: 0.613  loss_cls: 0.184  loss_box_reg: 0.355  loss_rpn_cls: 0.016  loss_rpn_loc: 0.041  time: 1.0439  data_tim
[08/01 12:32:46 d2.utils.events]:  eta: 0:20:56  iter: 299  total_loss: 0.680  loss_cls: 0.217  loss_box_reg: 0.359  loss_rpn_cls: 0.025  loss_rpn_loc: 0.057  time: 1.0440  data_tim
[08/01 12:33:08 d2.utils.events]:  eta: 0:20:39  iter: 319  total_loss: 0.599  loss_cls: 0.197  loss_box_reg: 0.363  loss_rpn_cls: 0.015  loss_rpn_loc: 0.043  time: 1.0470  data_tim
[08/01 12:33:29 d2.utils.events]:  eta: 0:20:18  iter: 339  total_loss: 0.551  loss_cls: 0.172  loss_box_reg: 0.324  loss_rpn_cls: 0.010  loss_rpn_loc: 0.035  time: 1.0468  data_tim
[08/01 12:33:50 d2.utils.events]:  eta: 0:19:57  iter: 359  total_loss: 0.543  loss_cls: 0.150  loss_box_reg: 0.309  loss_rpn_cls: 0.010  loss_rpn_loc: 0.050  time: 1.0467  data_tim
[08/01 12:34:12 d2.utils.events]:  eta: 0:19:38  iter: 379  total_loss: 0.565  loss_cls: 0.163  loss_box_reg: 0.295  loss_rpn_cls: 0.024  loss_rpn_loc: 0.037  time: 1.0480  data_tim
[08/01 12:34:31 d2.utils.events]:  eta: 0:19:14  iter: 399  total_loss: 0.620  loss_cls: 0.212  loss_box_reg: 0.326  loss_rpn_cls: 0.016  loss_rpn_loc: 0.036  time: 1.0448  data_tim
[08/01 12:34:53 d2.utils.events]:  eta: 0:18:54  iter: 419  total_loss: 0.574  loss_cls: 0.167  loss_box_reg: 0.323  loss_rpn_cls: 0.017  loss_rpn_loc: 0.043  time: 1.0460  data_tim
[08/01 12:35:14 d2.utils.events]:  eta: 0:18:34  iter: 439  total_loss: 0.561  loss_cls: 0.160  loss_box_reg: 0.301  loss_rpn_cls: 0.013  loss_rpn_loc: 0.040  time: 1.0468  data_tim
[08/01 12:35:34 d2.utils.events]:  eta: 0:18:10  iter: 459  total_loss: 0.543  loss_cls: 0.171  loss_box_reg: 0.332  loss_rpn_cls: 0.010  loss_rpn_loc: 0.034  time: 1.0440  data_tim
[08/01 12:35:54 d2.utils.events]:  eta: 0:17:48  iter: 479  total_loss: 0.532  loss_cls: 0.172  loss_box_reg: 0.329  loss_rpn_cls: 0.009  loss_rpn_loc: 0.038  time: 1.0438  data_tim
```

Training Process (Results from Colab)

Note that, here we also calculate the accuracy for every 500 iterations on the Validation Set.

Step 5: Inference using the Trained Model

It's time to infer the results by testing the model on the Validation Set.

An output folder gets saved in the local storage after successful completion of training in which the final weights are stored. You can save this folder for inferencing from this model in the future.

```python
from detectron2.utils.visualizer import ColorMode

#Use the final weights generated after successful training for inference
cfg.MODEL.WEIGHTS = os.path.join(cfg.OUTPUT_DIR, "model_final.pth")

cfg.MODEL.ROI_HEADS.SCORE_THRESH_TEST = 0.8  # set the testing threshold for this model
#Pass the validation dataset
cfg.DATASETS.TEST = ("boardetect_val", )

predictor = DefaultPredictor(cfg)

dataset_dicts = get_board_dicts("Text_Detection_Dataset_COCO_Format/val")
for d in random.sample(dataset_dicts, 3):
    im = cv2.imread(d["file_name"])
    outputs = predictor(im)
    v = Visualizer(im[:, :, ::-1],
                   metadata=board_metadata,
                   scale=0.8,
                   instance_mode=ColorMode.IMAGE
    )
    v = v.draw_instance_predictions(outputs["instances"].to("cpu")) #Passing the predict
    cv2_imshow(v.get_image()[:, :, ::-1])
```

predictions.py hosted with ❤️ by GitHub view raw

Results:

Step 6: Evaluation of the Trained Model

Usually, the model is evaluated following the COCO Standards of evaluation. Mean Average Precision (mAP) is used to evaluate the performance of the model. Here's an article that precisely gives an idea on the mAP.

```python
#import the COCO Evaluator to use the COCO Metrics
from detectron2.evaluation import COCOEvaluator, inference_on_dataset
from detectron2.data import build_detection_test_loader

#Call the COCO Evaluator function and pass the Validation Dataset
evaluator = COCOEvaluator("boardetect_val", cfg, False, output_dir="/output/")
val_loader = build_detection_test_loader(cfg, "boardetect_val")

#Use the created predicted model in the previous step
inference_on_dataset(predictor.model, val_loader, evaluator)
```

evaluation.py hosted with ❤️ by GitHub view raw

```
[07/28 12:59:29 d2.data.common]: Serializing 27 elements to byte tensors and concatenating them all ...
[07/28 12:59:29 d2.data.common]: Serialized dataset takes 0.01 MiB
[07/28 12:59:29 d2.evaluation.evaluator]: Start inference on 27 images
[07/28 12:59:33 d2.evaluation.evaluator]: Inference done 11/27. 0.3363 s / img. ETA=0:00:05
[07/28 12:59:39 d2.evaluation.evaluator]: Inference done 27/27. 0.3246 s / img. ETA=0:00:00
[07/28 12:59:39 d2.evaluation.evaluator]: Total inference time: 0:00:07.273351 (0.330607 s / img per device, on 1 devices)
[07/28 12:59:39 d2.evaluation.evaluator]: Total inference pure compute time: 0:00:07 (0.324569 s / img per device, on 1 devices)
[07/28 12:59:39 d2.evaluation.coco_evaluation]: Preparing results for COCO format ...
[07/28 12:59:39 d2.evaluation.coco_evaluation]: Saving results to /output/coco_instances_results.json
[07/28 12:59:39 d2.evaluation.coco_evaluation]: Evaluating predictions ...
Loading and preparing results...
DONE (t=0.00s)
creating index...
index created!
Running per image evaluation...
Evaluate annotation type *bbox*
DONE (t=0.07s).
Accumulating evaluation results...
DONE (t=0.03s).
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.486
 Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.794
 Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.588
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.416
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.541
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.454
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.195
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.536
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.538
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.498
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.610
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.519
[07/28 12:59:39 d2.evaluation.coco_evaluation]: Evaluation results for bbox:
|   AP   |  AP50  |  AP75  |  APs   |  APm   |  APl   |
|:------:|:------:|:------:|:------:|:------:|:------:|
| 48.614 | 79.410 | 58.818 | 41.596 | 54.075 | 45.368 |
[07/28 12:59:39 d2.evaluation.coco_evaluation]: Per-category bbox AP:
| category   | AP     | category   | AP     | category   | AP     |
|:-----------|:-------|:-----------|:-------|:-----------|:-------|
| HINDI      | 57.467 | ENGLISH    | 45.207 | OTHER      | 43.168 |
OrderedDict([('bbox',
              {'AP': 48.614130895033696,
               'AP-ENGLISH': 45.20707317037124,
```

Evaluation Metrics (Results from Colab)

We get an accuracy of around 79.4% for an IoU of 0.5 which is not that bad. It can be increased certainly by tweaking the parameters a bit and increasing the number of iterations. But keep an eye on training extensively as the model might overfit the training set.

Go through this Colab Notebook if you need inference from a saved model.

Conclusion

In this article, I emphasized on the procedure of objection detection using a custom dataset using detectron 2 rather than focusing on gaining higher accuracy.

Although this seems to be a pretty straight forward process, there's a lot to explore in Detectron 2's library. We have a good amount of optimization parameters that can be further tweaked to attain higher accuracy, which completely depends on one's custom dataset.

Hope you've learned something new today.

You can download the notebook from my Github repository and try running it on Google Colab or Jupyter Notebooks.

You could read more about Object Detection and the legacy R-CNN's in my previous article.

If you'd like to get in touch, connect with me on LinkedIn.

Artificial Intelligence Computer Vision Object Detection Deep Learning Towards Data Science