Economics 2355: Fine-Tuning Object Detection Models in Practice

Melissa Dell

Harvard University

February 2021

Outline

Selecting an Object Detection Model

Overview of Detectron2

How-to in D2

Recap

- In the last lecture, we covered the "R-CNN series" of object detection models: R-CNN, Fast R-CNN, and Faster R-CNN
- Later on, we also discussed Mask R-CNN, the instance segmenting successor to Faster R-CNN
- For all intents and purposes, each model in the main R-CNN series of object detection models strictly improved upon the last - we know this theoretically due to performance-motivated architectural adjustments, and empirically due to benchmarked results on object detection tasks

Recap

- However, as discussed, Mask R-CNN is essentially Faster R-CNN with a fully-connected branch to predict instance segmentation masks
- Does this mean we should be indifferent between Faster R-CNN and Mask R-CNN, or even prefer Faster R-CNN because it only does object detection?

Mask R-CNN is SOTA for object detection

- As it turns out, we should not be indifferent between Mask R-CNN and Faster R-CNN, and in fact have reason to prefer Mask R-CNN, which is now considered to be SOTA for *object detection* tasks

	backbone	AP^{bb}	AP^{bb}_{50}	AP^{bb}_{75}	AP^{bb}_{S}	AP^{bb}_{M}	AP^{bb}_{L}
Faster R-CNN+++ [19]	ResNet-101-C4	34.9	55.7	37.4	15.6	38.7	50.9
Faster R-CNN w FPN [27]	ResNet-101-FPN	36.2	59.1	39.0	18.2	39.0	48.2
Faster R-CNN by G-RMI [21]	Inception-ResNet-v2 [41]	34.7	55.5	36.7	13.5	38.1	52.0
Faster R-CNN w TDM [39]	Inception-ResNet-v2-TDM	36.8	57.7	39.2	16.2	39.8	**52.1**
Faster R-CNN, RoIAlign	ResNet-101-FPN	37.3	59.6	40.3	19.8	40.2	48.8
Mask R-CNN	ResNet-101-FPN	38.2	60.3	41.7	20.1	41.1	50.2
Mask R-CNN	ResNeXt-101-FPN	**39.8**	**62.3**	**43.4**	**22.1**	**43.2**	51.2

Table 3. **Object detection** *single-model* results (bounding box AP), *vs.* state-of-the-art on `test-dev`. Mask R-CNN using ResNet-101-FPN outperforms the base variants of all previous state-of-the-art models (the mask output is ignored in these experiments). The gains of Mask R-CNN over [27] come from using RoIAlign (+1.1 AP^{bb}), multitask training (+0.9 AP^{bb}), and ResNeXt-101 (+1.6 AP^{bb}).

- The benchmarked results speak for themselves, but why would we expect Mask R-CNN to do better than Faster R-CNN?

Mask R-CNN is SOTA for object detection

- The answer is the multi-task loss of the Mask R-CNN - experimentally, it can be shown that the mask-based instance segmentation loss Mask R-CNN is trained on (in addition to classification and bounding box losses) yields benefits for object detection as well
- As the authors state: "This gap of Mask R-CNN on box detection is therefore due solely to the benefits of multi-task training." (He et al., 2018)
- Intuitively, one could conjecture that really good bounding box predictions are a necessary precursor to really good instance mask predictions, i.e., the mask-based loss gives the model "greater incentive" to get better at predicting bounding boxes

Mask R-CNN is SOTA for object detection

- The takeaway: for most object detection purposes, you'll want to fine-tune a Mask R-CNN as your object detection model
- In our experience, layout analysis is no exception
- So, how does one fine-tune a Mask R-CNN (or Faster R-CNN) in practice?

Outline

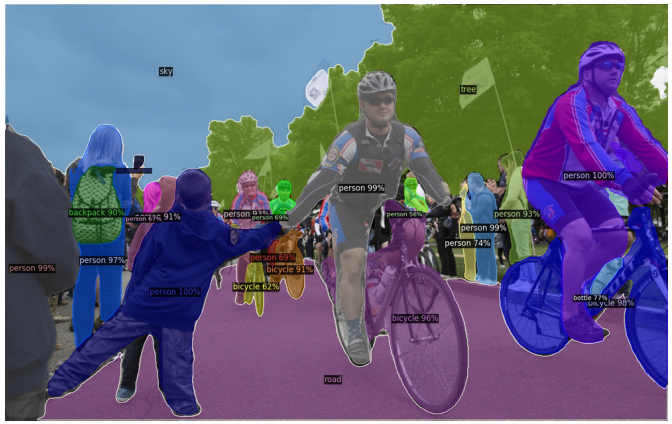
Selecting an Object Detection Model

Overview of Detectron2

How-to in D2

Introduction

- Detectron2 (D2) is a PyTorch fueled, Facebook AI Research (FAIR) authored "software system" that implements SOTA object detection models - practically, this means D2 is a codebase for object detection accessible via a GitHub repository: `https://github.com/facebookresearch/detectron2`
- D2 facilitates running more than just object detection models - it provides models and model modes for keypoint detection, semantic segmentation, instance segmentation, etc. - but object detection in D2 will be the focus of this lecture



Fundamentals

- D2 can be seen as a wrapper for a huge set of complex object detection model training and testing functions
- Therefore, at a very high level of abstraction, D2 is like a function that...
  - requires three inputs - (1) images, (2) a model checkpoint file (i.e., saved model weights from previous training), and (3) a model configuration file...
  - to produce an output (e.g., at training time, new model weights/checkpoints, and, at testing time, bounding boxes and classes for all objects detected in your image), or...
  - outputs = D2(images, checkpoint, config)
- As a reminder, when one talks about "fine-tuning," they are usually referring to the process of training a model from an existing checkpoint, or a model weight initialization arrived at through previous training

Installation

- For most purposes, you'll want to access D2 by first cloning the D2 GitHub repo to your machine, and then installing D2 from its source code; the former enables you to run various D2 scripts from the command line, and the latter gives you access to the D2 API in Python

```
# install D2 from source
git clone https://github.com/facebookresearch/detectron2.git
python -m pip install -e detectron2
```

Installation

- Alternatively, you may want to install D2 in a Docker container, for which FAIR has made a Docker image
  - For the upfront cost of learning a bit about Docker, you get a containerized (insulated) environment from which to execute D2, one in which dependency management for D2 is entirely taken care of
  - Docker containers can also make running apps or services with D2 much easier, as well as sharing a D2 project with collaborators
  - However, running Docker containers on Mac is notoriously memory burdensome, and even on other machines a container may leave a non-trivial memory footprint
  - Typically, for personal development and experimentation on a CPU, it is easiest to build D2 from source code in a fresh Python virtual environment

Installation

- FAIR also provides pre-built-from-source-code versions of D2 for Linux users, which make for a faster and less prone-to-error installation process, and for which you only need to know the version of PyTorch you have installed (and, if using a GPU, which CUDA version you have installed)
- For those unfamiliar, CUDA is the NVIDIA authored framework for general purpose GPU computation - it is necessary to have CUDA configured to use an NVIDIA GPU with PyTorch or for any other AI/ML application
  - Note: for those considering purchasing a GPU, NVIDIA GPUs are far and away the most well developed and supported GPUs for work in deep learning
- (See `INSTALL.md` in the D2 repo for more detail on installation best practices and troubleshooting)

Detectron2 from the command line

- D2 enables you to train and evaluate (test) models from the command line of Unix environments with the below syntax
- We will discuss the capitalized code at the bottom of the train command shortly; for now, note these are config file parameters being adjusted at run time
- (See `GETTING_STARTED.md` in the D2 repo for more on testing and training in the command line)

```
# train from command line
./train_net.py \
  --config-file /path/to/eg/mask_rcnn_R_50_FPN_1x.yaml \
  --num-gpus 1 \ # count of GPUs (for distributed training)
  SOLVER.IMS_PER_BATCH 2 SOLVER.BASE_LR 0.0025
# evaluate from command line
./train_net.py \
  --config-file /path/to/eg/mask_rcnn_R_50_FPN_1x.yaml \
  --eval-only MODEL.WEIGHTS /path/to/checkpoint_file
```

Detectron2 using the API

- When integrating D2 into a more expansive overall data processing pipeline, it is often necessary to call D2 functions and classes programmatically in Python
- A typical D2 workflow will involve monitoring and evaluating D2 training from the command line, after which D2 model inference is performed in Python
- For example...

Detectron2 using the API

```python
import json
import cv2
from detectron2.config import get_cfg
from detectron2.engine import DefaultPredictor

# specify config file
cfg = get_cfg() # get default config file
cfg.merge_from_file("path/to/eg/mask_rcnn_R_50_FPN_3x.yaml") # merge in custom params
cfg.MODEL.WEIGHTS = "path/to/model/checkpoint" # set checkpoint, if not already in yaml

# load image
im = cv2.imread("./input.jpg")

# run model inference via predictor class call
predictor = DefaultPredictor(cfg) # creates model from config, puts it in inference mode
outputs = predictor(im)

# print outputs
p_classes = outputs["instances"].pred_classes.tolist()
p_boxes = outputs["instances"].pred_boxes.tensor.tolist()
print(p_classes)
print(p_boxes)

# save outputs to json file
dict_for_json = {id: {'class': cls, 'box': box} for id, (cls, box) \
    in enumerate(zip(p_classes, p_boxes))} # put outputs in dict struct
with open('./out.json') as f:
    json.dump(dict_for_json, f)
```

Pre-trained models

- This lecture is about fine-tuning, not training from scratch, because in practice the people behind D2 - as well as other researchers - have made various pre-trained models (i.e., model checkpoint files) open source
- FAIR has trained both Faster and Mask R-CNN models with a variety of architectural modifications (e.g., FPN) for a variety of objection detection and recognition tasks and datasets, all of which can be found in D2's "Model Zoo"
- The COCO detection and instance segmentation pre-trained models available in the Model Zoo, which have been trained end-to-end to detect and segment objects in natural images, could be used as checkpoints for the purposes of fine-tuning an object detection model...

Pre-trained models

- ... However, for layout analysis, or any other detection task sufficiently dissimilar from natural images, in practice one tends to be most interested in initializing a model with just the ImageNet pre-trained backbones that D2 provides, e.g., ResNet, ResNeXt
  - These CNN backbones are very time and compute intensive to train from scratch, in contrast to the RPN and regression and classification heads of a given object detection model
  - Although it still might seem strange to use a backbone trained on natural images for layout analysis, recall that a CNN backbone acts as a sort of "feature extractor" for the model, and, especially in the early layers of the CNN, there are likely general purpose features being picked up that are shared between documents and natural images, e.g., edges and line segments
  - If trained end-to-end, the weights of the later CNN layers will also update in a way that makes them better feature extractors for layout analysis specific object detection tasks

Configurations

- D2 also provides a library of configuration files (or "config" files for short) that allow one to customize an object detection model from various presets - a given config file controls everything from choice of model architecture to the setting of model hyperparameters
- Config files have a particular naming convention in D2, which looks like the following: `objectDetectionModelArchitecture_backboneType_backboneDepth_backboneArchitecture_COCO.yaml`
- For example: `mask_rcnn_R_101_FPN_3x.yaml`
  - objectDetectionModelArchitecture = Mask R-CNN
  - backboneType = ResNet
  - backboneDepth = 101 layers
  - backboneArchitecture = FPN
  - COCO = reference to COCO training schedule - unimportant if just using pre-trained backbone
- Let's look at this config file...

Configurations

```
_BASE_: "../Base-RCNN-FPN.yaml"
MODEL:
  WEIGHTS: "detectron2://ImageNetPretrained/MSRA/R-101.pkl"
  MASK_ON: True
  RESNETS:
    DEPTH: 101
SOLVER:
  STEPS: (210000, 250000)
  MAX_ITER: 270000
```

- We see backbone weights being specified, as well as ResNet depth, and a path to another config file called `Base-RCNN-FPN.yaml`
- Let's take a look at the first few lines in this "base" config file

Configurations

```
MODEL:
  META_ARCHITECTURE: "GeneralizedRCNN"
  BACKBONE:
    NAME: "build_resnet_fpn_backbone"
  RESNETS:
    OUT_FEATURES: ["res2", "res3", "res4", "res5"]
  FPN:
    IN_FEATURES: ["res2", "res3", "res4", "res5"]
  ANCHOR_GENERATOR:
    SIZES: [[32], [64], [128], [256], [512]]
    ASPECT_RATIOS: [[0.5, 1.0, 2.0]]
```

- We see a "GeneralizedRCNN" architecture being specified, as well as anchor (box) hyperparameters, and selection of certain FPN feature maps, among other things - and this is only the first few lines of the file
- The capitalized code seen in the training command in the slide "Detectron2 from the command line" is modifying exactly the sort of capitalized code found in these config files

Configurations

- Just from these few config file snapshots, it can be seen that configuration files in D2 inherit config parameter values from one another
- `mask_rcnn_R_101_FPN_3x.yaml` inherits (and overwrites) the config parameters in `Base-RCNN-FPN.yaml`; furthermore, `Base-RCNN-FPN.yaml` inherits (and overwrites) the config parameters in the repo file found at `detectron2/config/defaults.py`
  - Moreover, any conifg parameter set in the training command overwrites its corresponding value in this hierarchy of inherited config parameters
- In fact, `detectron2/config/defaults.py` is the file that defines and describes every config parameter in D2 - it can serve as a one-stop-shop reference for setting up a D2 config file

Outline

Selecting an Object Detection Model

Overview of Detectron2

How-to in D2

How-to in D2

- At this point, we've covered a lot of the D2 basics, and just discussed the richness of config file parameters available for model customization
- We can now have a more nuanced discussion about how certain architectural choices, hyperparameter settings, and training and evaluation best practices can be implemented in D2 in practice - towards the ends of layout analysis, or anything else

Data pre-processing

- D2 implements data pre-processing in two ways: pixel value normalization and resizing of images
- Default values for pixel normalization, based on ImageNet, usually should not be changed manually
- Image resizing is multi-faceted
  - `MAX_SIZE_{TRAIN, TEST}` and `MIN_SIZE_{TRAIN, TEST}` parameters govern how the longer and shorter sides of an image, respectively, are scaled at train and test time
  - `MAX_SIZE_{TRAIN, TEST}` will simply cap the number of pixels in the longer side, scaling the image down when the max is surpassed
  - `MIN_SIZE_{TRAIN, TEST}` will scale the shorter side of the image down to the specified value (or set of values, which are selected across at random as a sort of low-cost data augmentation)

Data pre-processing

```
# defaults
MODEL.PIXEL_MEAN = [103.530, 116.280, 123.675]
MODEL.PIXEL_STD = [1.0, 1.0, 1.0]
INPUT.MIN_SIZE_TRAIN = (800,)
INPUT.MIN_SIZE_TRAIN_SAMPLING = "choice"
INPUT.MAX_SIZE_TRAIN = 1333
INPUT.MIN_SIZE_TEST = 800
INPUT.MAX_SIZE_TEST = 1333
```

- In practice, for layout analysis, which can easily feature hundreds of layout objects that need to be detected, the higher the resolution, the better the detection accuracy
- However, once resolution becomes high enough, one is liable to run out of GPU memory
- Moreover, the tradeoff for higher image resolution and the accuracy it yields is increased computation time - experimenting with various resolutions/image scaling parameters is therefore recommended for processing data at scale

Initialization, checkpoints, and transfer learning

- D2 models can be initialized from a given model checkpoint (weights from a previously trained model) by setting the config parameter `MODEL.WEIGHTS` to the path of an appropriate checkpoint file - this is all there is to transfer learning in D2
  - Valid model checkpoint files typically end in `.pth` or `.pkl`
- As mentioned, checkpoint files for, e.g., ImageNet backbones can be found in the D2 Model Zoo
- Checkpoint files can also be saved periodically as a part of the model training process; this is controlled by the parameter `SOLVER.CHECKPOINT_PERIOD`

Initialization, checkpoints, and transfer learning

- For those interested in layout analysis, PubLayNet trained model checkpoints have been made open source by a non-FAIR author at `https://github.com/hpanwar08/detectron2`
- In practice, we have found that initializing from a PubLayNet pre-trained model for the purposes of performing another document layout analysis task can cut training time by half (holding detection accuracy constant)

Optimization

- Optimization in D2 is controlled by "solver" hyperparameters; some of the most important and their default values are listed on the next slide
- In practice, `SOLVER.BASE_LR` (base learning rate), `SOLVER.MAX_ITER` (number of iterations the model runs/number of gradient updates the model makes), `SOLVER.STEPS` (list of iterations at which LR will be "stepped down" by factor of gamma), and `SOLVER.GAMMA` will be the most commonly tuned solver hyperparameters, varying distinctly use case by use case

Optimization

- In the D2 framework, one must keep track of training epochs oneself, as they are not defined explicitly - the default training sampler called by the default data loader creates an "infinite stream" of randomly shuffled data indices for the model to train on, i.e., `shuffle(range(size))+shuffle(range(size))+...`
- For a given training dataset, one can simply increase `SOLVER.MAX_ITER` to be equal to the desired number of epochs, given by `SOLVER.MAX_ITER * SOLVER.IMS_PER_BATCH / len(images)`

Optimization

- `SOLVER.IMS_PER_BATCH` - specifying the number of images per training mini-batch, to be split across all GPUs - is important to tune in conjunction with `SOLVER.BASE_LR`
- One should do so based on the "Linear Scaling Rule" introduced in Goyal et al. (2017): "when the minibatch size is multiplied by k, multiply the learning rate by k"

Optimization

- It is also recommended to include a "warmup" in your LR schedule, also per guidance from Goyal et al. (2017), to overcome some routinely empirically encountered optimization difficulties
- `SOLVER.WARMUP_METHOD` when set to "linear" will induce a warmup phase of the LR that lasts for the first `SOLVER.WARMUP_ITERS` training iterations, during which the LR will linearly grow from zero with a slope defined by `SOLVER.WARMUP_FACTOR`
- When `SOLVER.WARMUP_FACTOR` is set to the reciprocal of `SOLVER.WARMUP_ITERS`, the LR will linearly grow from zero to `SOLVER.BASE_LR` in the warmup phase; this is what D2 does by default

Optimization

```
# defaults
SOLVER.MAX_ITER = 40000
SOLVER.BASE_LR = 0.001
SOLVER.MOMENTUM = 0.9
SOLVER.NESTEROV = False
SOLVER.GAMMA = 0.1
SOLVER.STEPS = (30000,)
SOLVER.IMS_PER_BATCH = 16
SOLVER.LR_SCHEDULER_NAME = "WarmupMultiStepLR"
SOLVER.WARMUP_METHOD = "linear"
SOLVER.WARMUP_FACTOR = 1.0 / 1000
SOLVER.WARMUP_ITERS = 1000
```

Other Important Parameters

- Various other config parameters will also be important to tune for a wide array of layout analysis (and related) use cases, including:
  - `MODEL.ROI_HEADS.NUM_CLASSES` - defines the number of classes in your images
  - `MODEL.ANCHOR_GENERATOR.SIZES` - sets the anchor box areas for each FPN feature map
  - `MODEL.ANCHOR_GENERATOR.ASPECT_RATIOS` - sets the anchor box aspect ratios for each FPN feature map (can be important to change if detecting very horizontally or vertically slender objects)
  - `MODEL.RPN.{PRE, POST}_NMS_TOPK_{TRAIN, TEST}` - defines the number of proposals from the RPN sent to the ROI heads (e.g., classification and regression heads) (as a rule of thumb, there should be over 10x more region proposals than the average number of objects in an image)

Monitoring the Learning Process

- When training a model, D2 will by default print loss metrics every 20 iterations
  - For example: `[01/16 20:11:31] d2.utils.events INFO: eta: 9:16:43 iter: 179 total_loss: 1.889 loss_cls: 0.4358 loss_box_reg: 0.6458 loss_mask: 0.2161 loss_rpn_cls: 0.3312 loss_rpn_loc: 0.2558 time: 2.3690 data_time: 0.0046 lr: 0.00044955 max_mem: 6127M`
  - These printed statements make for quick and easy real time monitoring of converging or diverging loss
- When a model has finished running, the output folder will contain a `metrics.json` file which can be used to plot loss metrics (or some people will write a script to plot the loss in real time for easy monitoring)

Visualizing Outputs

- In order to sanity check an object detection model, it can be very helpful to visualize the results of model inference
- This is most easily accomplished in Python using the D2 API, as seen in the code on the next slide - one can quickly turn code like this into a script for visualizing model outputs

Visualizing Outputs

```
# assuming DefaultPredictor code from earlier is in the same file
from detectron2.utils.visualizer import Visualizer
from detectron2.data.catalog import Metadata

# create custom dataset metadata
metadata = Metadata(name="my_dataset", json_file=None,
    image_root="/path/to/image/dir", evaluator_type='coco',
    thing_colors=[(255, 0, 0), (255, 191, 0), (64, 255, 0)],
    thing_classes=["class1", "class2", "class3"],
    thing_dataset_id_to_contiguous_id={0:0, 1:1, 2:2}
)

# read in image and perform inference
im = cv2.imread("./input2.jpg")
outputs = predictor(im)

# visualize
v = Visualizer(im[:, :, ::-1],
   metadata=metadata,
   scale=0.5,
   instance_mode=ColorMode.SEGMENTATION
)
vis = v.draw_instance_predictions(outputs["instances"].to("cpu"))
cv2.imshow(vis.get_image()[:, :, ::-1])
```

Visualizing Outputs