Branch: master

Find file Copy path

pytorch-faster-rcnn / README.md

ruotianluo Update README.md

3e533c2 on 1 Nov 2019

10 contributors

Raw Blame History

294 lines (243 sloc) 19.9 KB

Notice(2019.11.2)

This repo was built back two years ago when there were no pytorch detection implementation that can achieve reasonable performance. At this time, there are many better repos out there, for example:

- detectron2
- mmdetection

Therefore, this repo will not be actively maintained.

Important notice:

If you used the master branch before Sep. 26 2017 and its corresponding pretrained model, **PLEASE PAY ATTENTION**: The old master branch in now under old_master, you can still run the code and download the pretrained model, but the pretrained model for that old master is not compatible to the current master!

The main differences between new and old master branch are in this two commits: 9d4c24e, c899ce7 The change is related to this issue; master now matches all the details in tf-faster-rcnn so that we can now convert pretrained tf model to pytorch model.

pytorch-faster-rcnn

A pytorch implementation of faster RCNN detection framework based on Xinlei Chen's tf-faster-rcnn. Xinlei Chen's repository is based on the python Caffe implementation of faster RCNN available here.

Note: Several minor modifications are made when reimplementing the framework, which give potential improvements. For details about the modifications and ablative analysis, please refer to the technical report An Implementation of Faster RCNN with Study for Region Sampling. If you are seeking to reproduce the results in the original paper, please use the official code or maybe the semi-official code. For details about the faster RCNN architecture please refer to the paper Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks.

Detection Performance

The current code supports **VGG16**, **Resnet V1** and **Mobilenet V1** models. We mainly tested it on plain VGG16 and Resnet101 architecture. As the baseline, we report numbers using a single model on a single convolution layer, so no multi-scale, no multi-stage bounding box regression, no skip-connection, no extra input is used. The only data augmentation technique is left-right flipping during training following the original Faster RCNN. All models are released.

With VGG16 (`conv5_3`):

- Train on VOC 2007 trainval and test on VOC 2007 test, **71.22**(from scratch) **70.75**(converted) (**70.8** for tf-faster-rcnn).
- Train on VOC 2007+2012 trainval and test on VOC 2007 test (R-FCN schedule), **75.33**(from scratch) **75.27**(converted) (**75.7** for tf-faster-rcnn).
- Train on COCO 2014 trainval35k and test on minival (900k/1190k) **29.2**(from scratch) **30.1**(converted) (**30.2** for tf-faster-rcnn).

With Resnet101 (last `conv4`):

- Train on VOC 2007 trainval and test on VOC 2007 test, **75.29**(from scratch) **75.76**(converted) (**75.7** for tf-faster-rcnn).
- Train on VOC 2007+2012 trainval and test on VOC 2007 test (R-FCN schedule), **79.26**(from scratch) **79.78**(converted) (**79.8** for tf-faster-rcnn).
- Train on COCO 2014 trainval35k and test on minival (800k/1190k), **35.1**(from scratch) **35.4**(converted) （**35.4** for tf-faster-rcnn).

More Results:

- Train Mobilenet (1.0, 224) on COCO 2014 trainval35k and test on minival (900k/1190k), **21.4**(from scratch), **21.9**(converted) （**21.8** for tf-faster-rcnn).
- Train Resnet50 on COCO 2014 trainval35k and test on minival (900k/1190k), **32.4**(converted) (**32.4** for tf-faster-rcnn).
- Train Resnet152 on COCO 2014 trainval35k and test on minival (900k/1190k), **36.7**(converted) (**36.1** for tf-faster-rcnn).

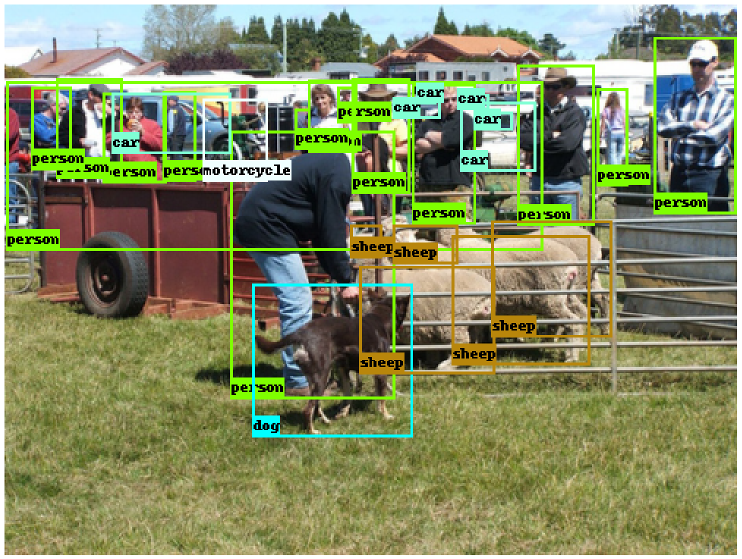
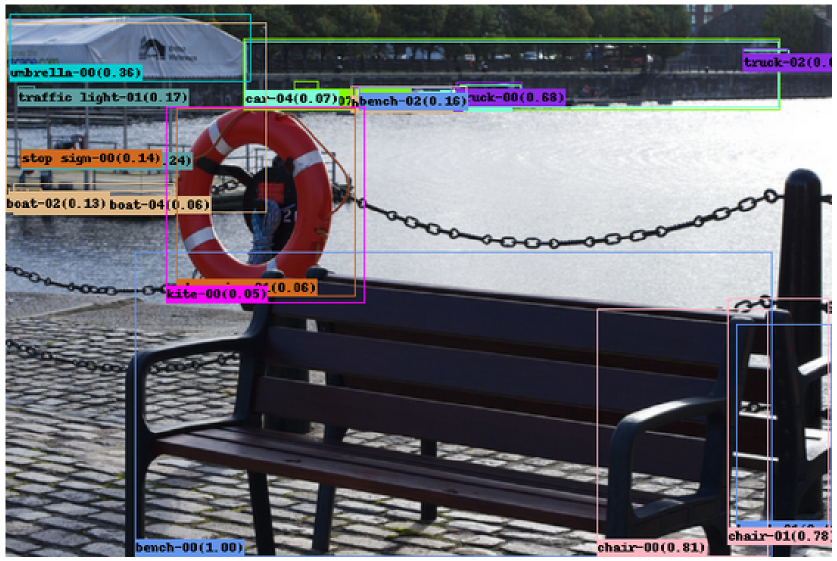
Approximate *baseline* setup from FPN (this repository does not contain training code for FPN yet):

- Train Resnet50 on COCO 2014 trainval35k and test on minival (900k/1190k), ~~**34.2**~~.
- Train Resnet101 on COCO 2014 trainval35k and test on minival (900k/1190k), ~~**37.4**~~.
- Train Resnet152 on COCO 2014 trainval35k and test on minival (900k/1190k), ~~**38.2**~~.

Note:

- Due to the randomness in GPU training especially for VOC, the best numbers are reported (with 2-3 attempts) here. According to Xinlei's experience, for COCO you can almost always get a very close number (within ~0.2%) despite the randomness.

- The numbers are obtained with the **default** testing scheme which selects region proposals using non-maximal suppression (TEST.MODE nms), the alternative testing scheme (TEST.MODE top) will likely result in slightly better performance (see report, for COCO it boosts 0.X AP).
- Since we keep the small proposals (< 16 pixels width/height), our performance is especially good for small objects.
- We do not set a threshold (instead of 0.05) for a detection to be included in the final result, which increases recall.
- Weight decay is set to 1e-4.
- For other minor modifications, please check the report. Notable ones include using `crop_and_resize` , and excluding ground truth boxes in RoIs during training.
- For COCO, we find the performance improving with more iterations, and potentially better performance can be achieved with even more iterations.
- For Resnets, we fix the first block (total 4) when fine-tuning the network, and only use `crop_and_resize` to resize the RoIs (7x7) without max-pool (which Xinlei finds useless especially for COCO). The final feature maps are average-pooled for classification and regression. All batch normalization parameters are fixed. Learning rate for biases is not doubled.
- For Mobilenets, we fix the first five layers when fine-tuning the network. All batch normalization parameters are fixed. Weight decay for Mobilenet layers is set to 4e-5.
- For approximate FPN baseline setup we simply resize the image with 800 pixels, add 32^2 anchors, and take 1000 proposals during testing.
- Check out here/here/here for the latest models, including longer COCO VGG16 models and Resnet ones.

	
Displayed Ground Truth on Tensorboard	Displayed Predictions on Tensorboard

Additional features

Additional features not mentioned in the report are added to make research life easier:

- **Support for train-and-validation**. During training, the validation data will also be tested from time to time to monitor the process and check potential overfitting. Ideally training and validation should be separate, where the model is loaded every time to test on validation. However Xinlei have implemented it in a joint way to save time and GPU memory. Though in the default setup the testing data is used for validation, no special attempts is made to overfit on testing set.
- **Support for resuming training**. Xinlei tried to store as much information as possible when snapshoting, with the purpose to resume training from the latest snapshot properly. The meta information includes current image index, permutation of images, and random state of numpy. However, when you resume training the random seed for tensorflow will be reset (not sure how to save the random state of tensorflow now), so it will result in a difference. **Note** that, the current implementation still cannot force the model to behave deterministically even with the random seeds set. Suggestion/solution is welcome and much appreciated.

- **Support for visualization**. The current implementation will summarize ground truth boxes, statistics of losses, activations and variables during training, and dump it to a separate folder for tensorboard visualization. The computing graph is also saved for debugging.

Prerequisites

- A basic pytorch installation. The code follows **1.0**. If you are using old **0.1.12** or **0.2** or **0.3** or **0.4**, you can checkout the corresponding branch.
- Torchvision **0.3**. This code uses `torchvision.ops` for `nms` , `roi_pool` and `roi_align`
- Python packages you might not have: `opencv-python` , `easydict` (similar to py-faster-rcnn). For `easydict` make sure you have the right version. Xinlei uses 1.6.
- tensorboard-pytorch to visualize the training and validation curve. Please build from source to use the latest tensorflow-tensorboard.
- ~~Docker users: Since the recent upgrade, the docker image on docker hub (https://hub.docker.com/r/mbuckler/tf-faster-rcnn-deps/) is no longer valid. However, you can still build your own image by using dockerfile located at `docker` folder (cuda 8 version, as it is required by Tensorflow r1.0.) And make sure following Tensorflow installation to install and use nvidia-docker[https://github.com/NVIDIA/nvidia-docker]. Last, after launching the container, you have to build the Cython modules within the running container.~~

Installation

1. Clone the repository

```
git clone https://github.com/ruotianluo/pytorch-faster-rcnn.git
```

2. Install the Python COCO API. The code requires the API to access COCO dataset.

```
cd data
git clone https://github.com/pdollar/coco.git
cd coco/PythonAPI
```

```
make
cd ../../..
```

Setup data

Please follow the instructions of py-faster-rcnn here to setup VOC and COCO datasets (Part of COCO is done). The steps involve downloading data and optionally creating soft links in the `data` folder. Since faster RCNN does not rely on pre-computed proposals, it is safe to ignore the steps that setup proposals.

If you find it useful, the `data/cache` folder created on Xinlei's side is also shared here.

Demo and Test with pre-trained models

1. Download pre-trained model (only google drive works)

- ~~Another server here.~~
- Google drive here.

(Optional) Instead of downloading my pretrained or converted model, you can also convert from tf-faster-rcnn model. You can download the tensorflow pretrained model from tf-faster-rcnn. Then run:

```
python tools/convert_from_tensorflow.py --tensorflow_model resnet_model.ckpt
python tools/convert_from_tensorflow_vgg.py --tensorflow_model vgg_model.ckpt
```

This script will create a `.pth` file with the same name in the same folder as the tensorflow model.

2. Create a folder and a soft link to use the pre-trained model

```
NET=res101
TRAIN_IMDB=voc_2007_trainval+voc_2012_trainval
mkdir -p output/${NET}/${TRAIN_IMDB}
cd output/${NET}/${TRAIN_IMDB}
```

```
ln -s ../../../data/voc_2007_trainval+voc_2012_trainval ./default
cd ../../..
```

3. Demo for testing on custom images

```
# at repository root
GPU_ID=0
CUDA_VISIBLE_DEVICES=${GPU_ID} ./tools/demo.py
```

Note: Resnet101 testing probably requires several gigabytes of memory, so if you encounter memory capacity issues, please install it with CPU support only. Refer to Issue 25.

4. Test with pre-trained Resnet101 models

```
GPU_ID=0
./experiments/scripts/test_faster_rcnn.sh $GPU_ID pascal_voc_0712 res101
```

Note: If you cannot get the reported numbers (79.8 on my side), then probably the NMS function is compiled improperly, refer to Issue 5.

Train your own model

1. Download pre-trained models and weights. The current code support VGG16 and Resnet V1 models. Pre-trained models are provided by pytorch-vgg and pytorch-resnet (the ones with caffe in the name), you can download the pre-trained models and set them in the `data/imagenet_weights` folder. For example for VGG16 model, you can set up like:

   ```
   mkdir -p data/imagenet_weights
   cd data/imagenet_weights
   python # open python in terminal and run the following Python code
   ```

```python
import torch
from torch.utils.model_zoo import load_url
from torchvision import models

sd = load_url("https://s3-us-west-2.amazonaws.com/jcjohns-models/vgg16-00b39a1b.pth")
sd['classifier.0.weight'] = sd['classifier.1.weight']
sd['classifier.0.bias'] = sd['classifier.1.bias']
del sd['classifier.1.weight']
del sd['classifier.1.bias']

sd['classifier.3.weight'] = sd['classifier.4.weight']
sd['classifier.3.bias'] = sd['classifier.4.bias']
del sd['classifier.4.weight']
del sd['classifier.4.bias']

torch.save(sd, "vgg16.pth")
```

```
cd ../..
```

For Resnet101, you can set up like:

```
mkdir -p data/imagenet_weights
cd data/imagenet_weights
# download from my gdrive (link in pytorch-resnet)
mv resnet101-caffe.pth res101.pth
cd ../..
```

For Mobilenet V1, you can set up like:

```
mkdir -p data/imagenet_weights
cd data/imagenet_weights
# download from my gdrive (https://drive.google.com/open?id=0B7fNdx_jAqhtZGJvZlpVeDhUN1k)
```

```
    mv mobilenet_v1_1.0_224.pth.pth mobile.pth
    cd ../..
```

2. Train (and test, evaluation)

```
./experiments/scripts/train_faster_rcnn.sh [GPU_ID] [DATASET] [NET]
# GPU_ID is the GPU you want to test on
# NET in {vgg16, res50, res101, res152} is the network arch to use
# DATASET {pascal_voc, pascal_voc_0712, coco} is defined in train_faster_rcnn.sh
# Examples:
./experiments/scripts/train_faster_rcnn.sh 0 pascal_voc vgg16
./experiments/scripts/train_faster_rcnn.sh 1 coco res101
```

Note: Please double check you have deleted soft link to the pre-trained models before training. If you find NaNs during training, please refer to Issue 86. Also if you want to have multi-gpu support, check out Issue 121.

3. Visualization with Tensorboard

```
tensorboard --logdir=tensorboard/vgg16/voc_2007_trainval/ --port=7001 &
tensorboard --logdir=tensorboard/vgg16/coco_2014_train+coco_2014_valminusminival/ --port=7002 &
```

4. Test and evaluate

```
./experiments/scripts/test_faster_rcnn.sh [GPU_ID] [DATASET] [NET]
# GPU_ID is the GPU you want to test on
# NET in {vgg16, res50, res101, res152} is the network arch to use
# DATASET {pascal_voc, pascal_voc_0712, coco} is defined in test_faster_rcnn.sh
# Examples:
./experiments/scripts/test_faster_rcnn.sh 0 pascal_voc vgg16
./experiments/scripts/test_faster_rcnn.sh 1 coco res101
```

5. You can use `tools/reval.sh` for re-evaluation

By default, trained networks are saved under:

```
output/[NET]/[DATASET]/default/
```

Test outputs are saved under:

```
output/[NET]/[DATASET]/default/[SNAPSHOT]/
```

Tensorboard information for train and validation is saved under:

```
tensorboard/[NET]/[DATASET]/default/
tensorboard/[NET]/[DATASET]/default_val/
```

The default number of training iterations is kept the same to the original faster RCNN for VOC 2007, however Xinlei finds it is beneficial to train longer (see report for COCO), probably due to the fact that the image batch size is one. For VOC 07+12 we switch to a 80k/110k schedule following R-FCN. Also note that due to the nondeterministic nature of the current implementation, the performance can vary a bit, but in general it should be within ~1% of the reported numbers for VOC, and ~0.2% of the reported numbers for COCO. Suggestions/Contributions are welcome.

Citation

If you find this implementation or the analysis conducted in our report helpful, please consider citing:

```
@article{chen17implementation,
    Author = {Xinlei Chen and Abhinav Gupta},
    Title = {An Implementation of Faster RCNN with Study for Region Sampling},
    Journal = {arXiv preprint arXiv:1702.02138},
    Year = {2017}
}
```

For convenience, here is the faster RCNN citation:

```
@inproceedings{renNIPS15fasterrcnn,
    Author = {Shaoqing Ren and Kaiming He and Ross Girshick and Jian Sun},
    Title = {Faster {R-CNN}: Towards Real-Time Object Detection
             with Region Proposal Networks},
    Booktitle = {Advances in Neural Information Processing Systems ({NIPS})},
    Year = {2015}
}
```

~~Detailed numbers from COCO server~~ (not supported)

All the models are trained on COCO 2014 trainval35k.

VGG16 COCO 2015 test-dev (900k/1190k):

```
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.297
 Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.504
 Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.312
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.128
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.325
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.421
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.272
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.399
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.409
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.187
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.451
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.591
```

VGG16 COCO 2015 test-std (900k/1190k):

```
Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.295
Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.501
Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.312
```

```
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.119
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.327
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.418
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.273
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.400
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.409
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.179
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.455
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.586
```