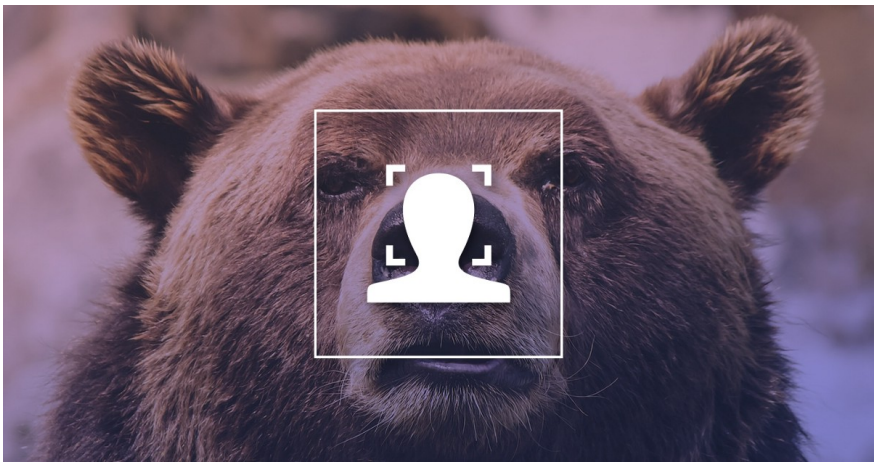
ed value.

May 17, 2017 · 7 min read

Modern Face Detection based on Deep Learning using Python and Mxnet



Modern Face Detection based on Deep Learning using Python and Mxnet by Wassa

In this post, we'll discuss and illustrate a fast and robust method for face detection using Python and Mxnet. At Wassa, some of our products rely on face detection. For example, in Facelytics, it is used for face attribute extraction like gender, age and so. It can also be used for face recognition, emotion recognition, some augmented reality applications or people counting. Depending on the purpose, robust real-time face detector may be needed. Taking into account people counting, it may require a fast and robust detector which can detect faces from different angles… We define two kinds of applications that need almost opposite qualities. Counting people needs a fast and robust detector which can detect faces in different conditions. On the contrary, attributes extraction hasn't a good accuracy if the detected face offers too much angle. In the following, we will show a robust and fast face detection that can be used for counting people in crowd.

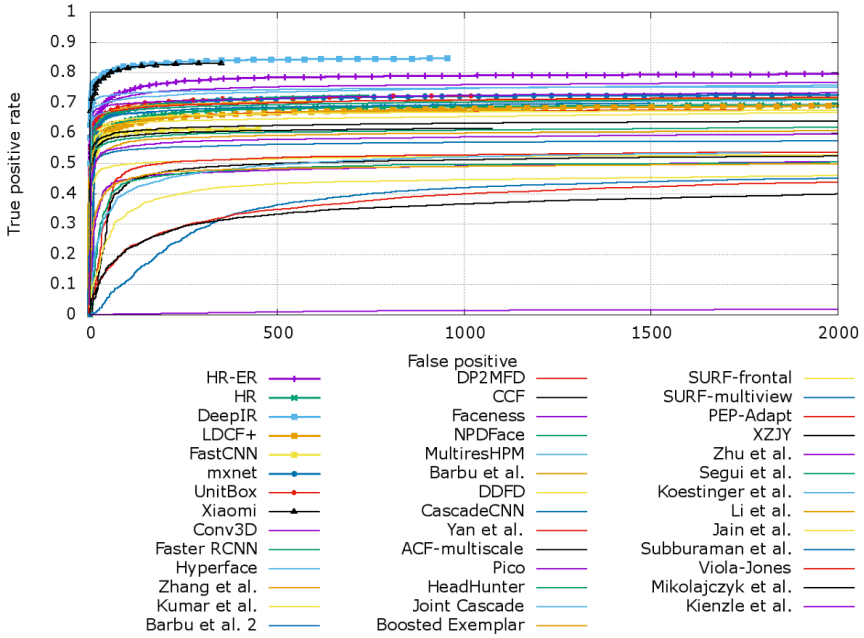
Mxnet

Mxnet is a flexible and efficient framework for deep learning applications. It has interface in lots of languages for execution and

training (C/C++, Python, R, etc…). It also supports a compact and limited API allowing only execution for embedded application on smart devices. The community is active and friendly. Like Tensorflow supported by Google or Torch supported by Facebook, Mxnet is supported by Baidu and recently by Amazon.

State of the art

One of the most used method for face detection has been the Viola Jones method for many years. This detection uses a cascade of haar classifiers. An advantage of this method is its implementation in OpenCV. Thanks to its availability, it is included in various tutorials demonstrating face detection. This method has been the reference in face detection since 2001 but computer vision has evolved a lot since then, and there are new methods which outperform the Viola Jones algorithm. An accurate catalog of these methods is listed by FDDB (the Face Detection Data Set and Benchmark). On this list, a large part of top algorithms use deep learning method. Some of them are accurate and relatively slow and other try to be as fast as possible.



In this tutorial, we will use a Mxnet implementation of the MTCNN algorithm designed by Zhang. This implementation can be found on the GitHub link.

MTCNN

The MTCNN algorithm works in three steps and use one neural network for each. The first part is a proposal network. It will predict potential face positions and their bounding boxes like an attention network in Faster R-CNN. The result of this step is a large number of face detections and lots of false detections. The second part uses images and outputs of the first prediction. It makes a refinement of the result to eliminate most of false detections and aggregate bounding boxes. The last part refines even more the predictions and adds facial landmarks predictions (in the original MTCNN implementation).

Requirements

The installation step can be the main issue of lots of these deep learning codes. To run our example, we need to install OpenCV, Mxnet and their binding in python3.

- Explanation for OpenCV installation link
- Explanation for Mxnet installation link

Docker Installation

If you can't install Mxnet/OpenCV on your computer or if you fail to install them and if you know Docker, we have created Docker Images with OpenCV and Mxnet installed.

First, if you don't already have Docker installed, the official method is easy to use and well documented (link).

The ***second*** step is to clone our repo with some DockerFiles that will automatically generate the right Docker image. This is my personal cheat sheet for OpenCV and other libraries installation.

In your working directory, insert the following command:

```
git clone https://github.com/edmBernard/DockerFiles
cd DockerFiles
make mxnet
```

These lines will download our DockerFiles and generate Docker images containing python3, OpenCV and Mxnet.

Third step, creates the container and enters in it. Go in your working directory and enter the following command:

```
sudo docker run -it --name tuto_face_detection -v
${PWD}:/home/dev/host mxnet:latest
```

If the installation worked correctly, you will obtain a command like this one:

```
root@d88bd1bfe1af:~/host#
```

and if you type `ls` you will see your working directory that only contain the DockerFiles folder for the moment:

```
root@d88bd1bfe1af:~/host# ls
DockerFiles
```

Small Docker cheat sheet

- Open a terminal in the docker: `sudo docker exec -it tuto_face_detection /bin/bash`
- Leave the docker: `exit`
- Delete the docker: `sudo docker rm -f tuto_face_detection`

Check Your Installation

Type these following commands inside the container to check if the installation worked. It will show Mxnet and OpenCV versions:

```
python3 -c 'import cv2; print(cv2.__version__)'
#3.2.0-dev
python3 -c 'import mxnet as mx; print(mx.__version__)'
#0.9.5
```

Right now, when this article was written (9th May 2017), we got Mxnet 0.9.5 and OpenCV 3.2.0. Newer versions will be fine too. We didn't test this code with older versions. OpenCV version upper than 3.0 may work. For Mxnet, it's more complex. It's an active framework and differences between versions can break some functionalities, mainly operators that didn't exist before. The API we use has changed in version 0.9.3 so older versions will not work.

If you don't get any errors, the installation should be fine and you can move forward.

MTCNN download

The Mxnet implementation of MTCNN comes with pre-trained models. We get them by cloning their GitHub repository. For this tutorial and to keep using python 3 instead of python 2 because of this, we fork the *Xuan Lin* repository to switch the python compatibility of the prediction. From the container, type the following command:

```
cd /home/dev/host # your working directory
git clone https://github.com/edmbernard/mtcnn.git
cd mtcnn
```

You must now have the following tree:

```
/home/dev/host
 ├── DockerFiles
 │ ├── cpuBoth show equivalent calculation time.
 │ ├── gpu
 │ ├── Makefile
 │ ├── python36
 │ └── README.md
 └── mtcnn
     ├── core
     ├── example
```

```
        ├── model
        ...
        ├── demo_on_stream_blog.py
        ├── demo_on_stream_blog_with_show.py
        ├── demo_on_stream.py
        ├── demo.py
        ...
```

The Code

The code complexity for running prediction with neural network is low in most cases. In fact, it depends if all the processes are run through the network or if we need a part outside. For example, the implementation of image classification using VGG16 needs only two lines, one for loading the model, one to pass the image in the model. The MTCNN algorithm combines 3 neural networks, a large part of the logic is in this concatenation. But thanks to the author, the code is clear and usable almost as it is. All the code, for this illustration is available here.

Import

```
# numpy: images and matrices computation
import numpy as np
# mxnet: deep learning
import mxnet as mx
# argparse: command line parsing
import argparse
# cv2: opencv image processing
import cv2
import time
# symbol: define the network structure
from core.symbol import P_Net, R_Net, O_Net
# detector: bind weight with structure and create a de
from core.detector import Detector
```

Default parameters

In this example, we'll use default parameters as follow.

```
def test_net(prefix=['model/pnet', 'model/rnet', 'model
```

- `ctx=mx.cpu(0)` : defines the execution on cpu (`mx.gpu(0)` to run on GPU)
- `camera_path='0'` : defines the camera we use. In this tutorial, we test with the integrated laptop webcam (`0`) and with an IP Camera Zavio D6320 (`rtsp://user:password@192.168.X.X:554/video.pro1`).

Load model

In Mxnet, the model is usually defined by two files, one with `.params` extension and one with `.json` extension. The `params` file contains all weights of the network and the `json` contains the layer structure of the network. This second file is optional if we recreate the structure in python code.

The MTCNN is composed by three different models named `P_Net` , `R_Net` , `O_Net` . The structure is defined by `P_Net()` , `R_Net()` , `O_Net()` functions and weights are saved in `pnet-0016.params` , `rnet-0016.params` , `onet-0016.params` files. The initialization part consists in loading the weights and binding weights in the structure. The model is loaded with the function `load_param()` . The filename is formed by the concatenation of prefix parameter (ex: `rnet`) and epoch parameter (ex: `16`). In Mxnet, almost every function takes the `ctx` (*context*) parameter. It defines on which device we run the code (CPU, GPU).

```
# load pnet model
args, auxs = load_param(prefix[0], epoch[0], convert=T
PNet = FcnDetector(P_Net("test"), ctx, args, auxs)

# load rnet model
args, auxs = load_param(prefix[1], epoch[0], convert=T
RNet = Detector(R_Net("test"), 24, batch_size[1], ctx,

```

Create detector

The neural network concatenation is done by the function `mtcnn_detector()` . It creates detectors from raw network.

```
mtcnn_detector = MtcnnDetector(detectors=detectors, ctx
                               stride=stride, threshold
```

Now all the deep learning part is set up. We still need to add the code to feed images in the face detector.

Processing code

```
try:
        capture = cv2.VideoCapture(int(camera_path))
    except ValueError as e:
        capture = cv2.VideoCapture(camera_path)

    try:
        first_loop = True
        while (capture.isOpened()):
            ret, img = capture.read()
            if img is None:
                continue

            # Initialize video writing
            if (first_loop):
                first_loop = False
                fourcc = cv2.VideoWriter_fourcc(*'H264
                h, w = img.shape[:2]
                writer = cv2.VideoWriter('test.mkv', f

            t1 = time.time()

            boxes, boxes_c = mtcnn_detector.detect_pne
            boxes, boxes_c = mtcnn_detector.detect_rne
            boxes, boxes_c = mtcnn_detector.detect_one

```

If you use the Docker version, you can't use windows output so you'll only write the result in a file. If you don't work in a Docker you can try the `demo_on_stream_blog_with_show.py` that will show in real time the annotated videos. The final code can be found here or here.

Execution

The code can be run from python REPL:

```
In [1]: import mxnet as mx
In [2]: from demo_on_stream_blog import test_net
In [3]: test_net(camera_path='0', ctx=mx.cpu(0))
```

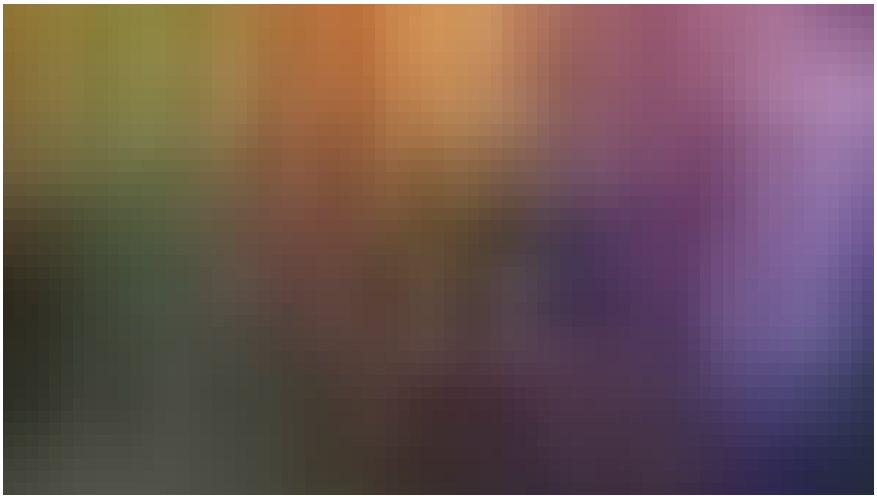
or from the command line if you got the complete code here with:

```
python3 demo_on_stream_blog — camera-path 0 — gpu -1
```

The processed video is named `test.mkv` in your working directory. We did tests on several devices with/without optimization on CPU and on GPU.

- with an i7 7700 and without particular optimization, we have a process at 2 FPS.
- with an i7 7500U and NNPACK optimization, we got around 5 FPS.
- and with an i7 7700 and with a basic graphic card GTX 1070 and CUDNN v6, we run at 40 FPS.

We tested with both webcam and IP camera. Both show equivalent calculation time. This face detection is fast with 2 frames processed per second without any optimization. And process streams near real time with NNPACK or real time with a GPU.



Face Detection demo by Wassa

Conclusion

The goal was to show that a deep learning algorithm can be fast and accurate even on CPU. With this example, we got a face detection in real-time that works on CPU and GPU. This method offers good performances and is one of the best face detection on FDDB. Deep learning can be easy to use and powerful.

Do you want to know more about Wassa?

Find us on:

- Facebook and Twitter
- LinkedIn
- GitHub
- our web site