Wonder Balls challenge by Yotam Ben-Shoshan

Given 15 images that contains a wonder ball develop a model that would detect and mark, with a purple circle, all the wonder balls in any given image. A good processing time would be under 0.8[sec] per image

My solution:

1. Heavily augment the images to pursue a bigger wonder ball data set.
2. Use a Single Shot Detector (SSD) neural net architecture to do transfer learning with the new wonder balls data set.

Step 1- Data handling:

Data augmentation

The most important thing for data driven model is the data itself therefore the first thing I did was to label the 15 images using VIA [1] tool and use those annotated image to generate a lager data set by using the data augmentation imagu library [2].The library supports a wide range of augmentation techniques and augments the bounding boxes as well.

In my work I augmented the data by using a mild affine transform, adding gaussian noise, flipping and cropping the image. augmentData.py is the code I wrote to do that.

In The end of the augmentation process I ended up with 750 augmented images + 15 original images. See figure 1 for some augmentation samples.

Figure 1

Image Resize

The given images are all 640x480 pixel size. The Neural Net architecture, that I choose, the input image is required to be 512x512 pixels. Therefore, to be able to feed the NN with the given data I choose to pad with zeros all the images to the size of 640x640 and then resize is it to 512x512. The reason I did that is to preserve the aspect ratio of the original images especially because the task is to detect balls therefore, I would like to preserve the shape feature. The resize code could be found in resizeImgs.py.

Step 2 – Train object detection Neural Net

Which architecture to use?

At the beginning of the project I was contemplating between 3 different object detection NN architectures RCNN, SSD and Yolo. According to an object detection guide I read [3] the SSD architecture should give a good tradeoff between accuracy and speed comparing to the other mentioned architectures see figure 2 (below):

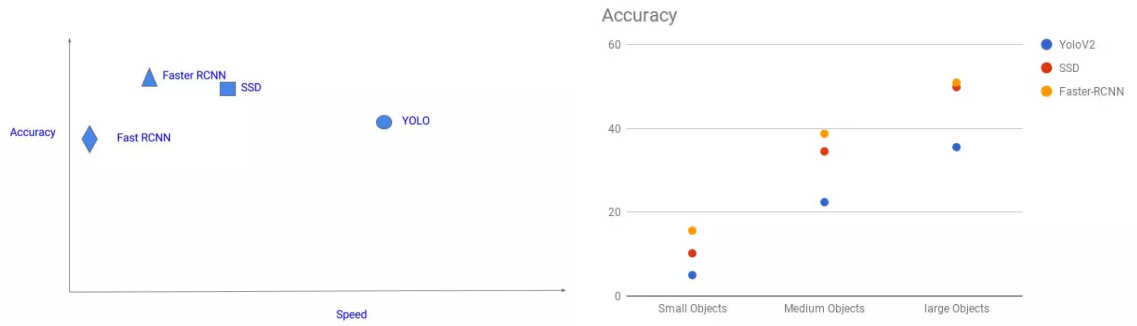


Figure 2

What Is a Single shot detector (SSD)?

In general, SSD is a deep neural network object detection approach that discretizes the output space of bounding boxes into a set of default boxes over different aspect ratios and scales per feature map location. At prediction time, the network generates scores for the presence of each object category in each default box and produces adjustments to the box to better match the object shape. Additionally, the network combines predictions from multiple feature maps with different resolutions to naturally handle objects of various sizes.

SSD code

I downloaded all the SSD code from git hub [4]. In SSD code there where 2 NN I could choose from: SSD300 and SSD512. I choose to work with the SSD512 because I wanted to be as closer to the original images size. In case of tight time constrains it could be a good option to try the SSD300 instead.

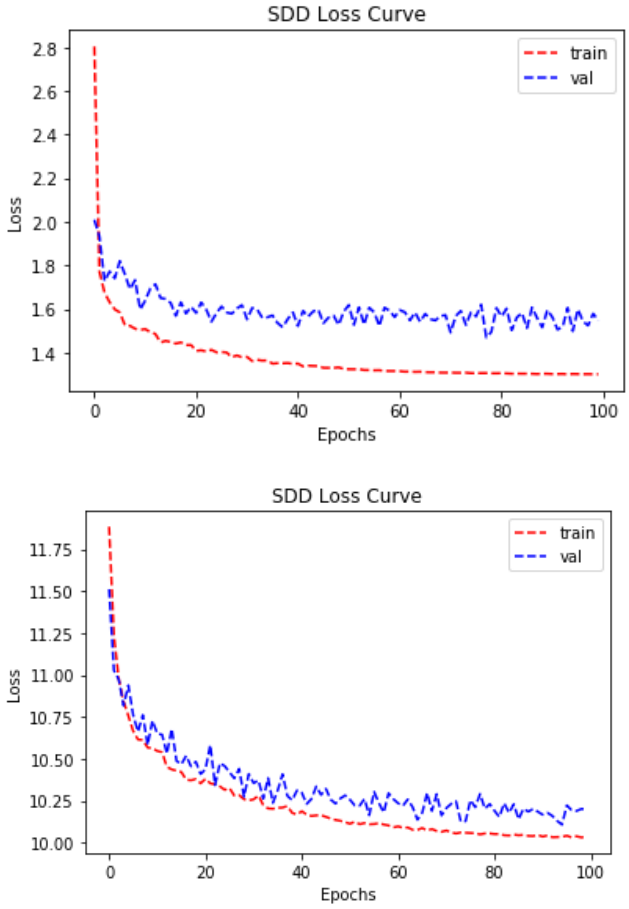
ssd512_training_transfe_ learning.ipnb – this Jupyter notebook was created based on the SSD300_traing.ipnb. In general, the code *Builds the model, generates data, train the model and evaluates it at the end.*

weight_sampling_tutorial.ipynb – This jupyter notebook is used to down sample the number of classes of the original NN, which had 80 classes to fit the new NN which has different number of classes.

Training the model

After running the train section in ssd512_training_transfer_ learning.ipnb I got figure 3 (A) graph train results. After ~40 epochs the validation loss had stopped descending while the train loss kept going down. That is an indication for overfitting of the model from epoch ~40. In order to avoid overfitting, I

increased the regularization parameter from 0.0005 to 0.0055. Figure 3 (B) is the train result after the regularization change Which in my opinion shows an improvement comparing to graph (A) because the difference between the validation error to the train error is smaller.

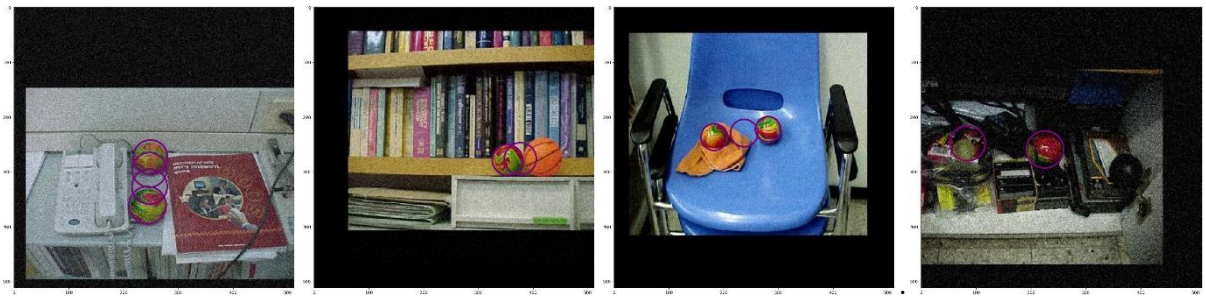


(A) *(B)*

Figure 3

Testing the model performance

In order to test the model performance, I generated output image for all the validation set (it was better if had prepared a test set), which was 153 images, and I started examining the results. In general, most images showed good inferenced results. The common mistakes where multiple detection of the same ball see figure 4 (A), (B) and (c) examples. The only false alarm, I found, is shown in figure 4 (D)



(A) (B) (C) (D)

Figure 4

In order to get rid of the multiple detection problem, I lowered the “iou_threshold” from 0.45 to 0.05. As consequence fewer bounding boxes overlap due to the new policy which does not allow bounding boxes to overlap if they have more than 5% percent overlap.

In order to avoid false alarm I change the “confidence_thresh” from 0.5 to 0.99 .

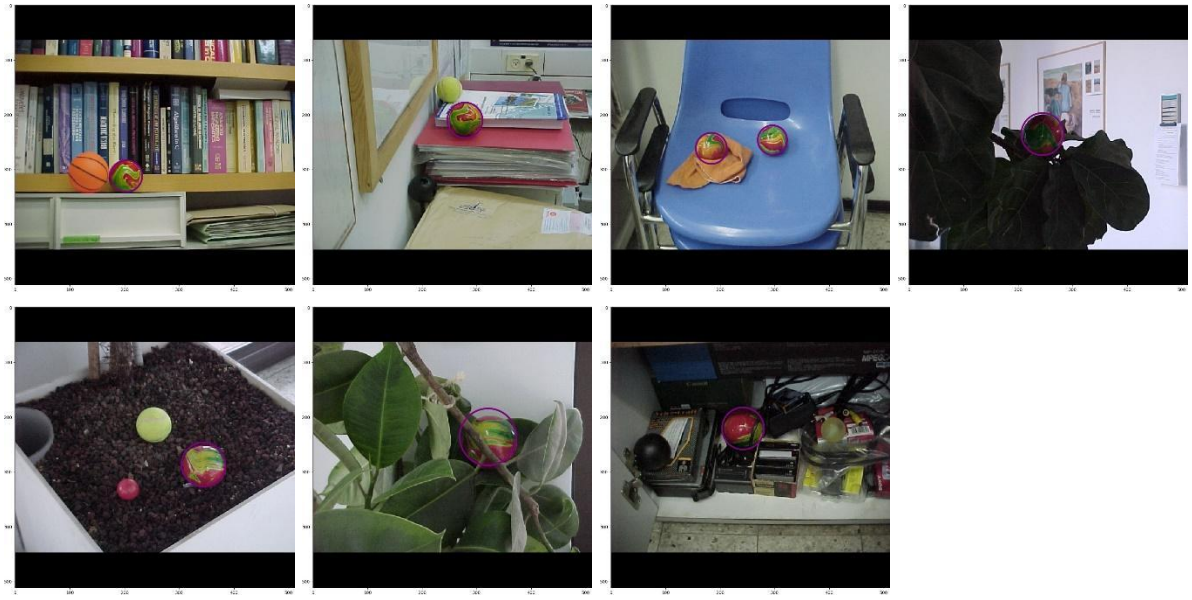
After applying those two changes the model has shown big improvement.

The last thing I did to evaluate the performance is to run a mAP (mean average precision) script which gave me result of 0.662.

Processing time:

On CPU	On Nvidia GPU m4000:
file ../examples\MVC-001F_resized.jpg process took: 1.3579583984891905[sec]	0.1141919000000371[sec]
file ../examples\MVC-002F_resized.jpg process took: 0.9892451597720588[sec]	0.13335429999995085[sec]
file ../examples\MVC-003F_resized.jpg process took: 0.9976932443914643[sec]	0.09998989999996866[sec]
file ../examples\MVC-004F_resized.jpg process took: 1.0349645940689287[sec]	0.15166659999999865[sec]
file ../examples\MVC-005F_resized.jpg process took: 0.9956346492067301[sec]	0.1248496000000614[sec]
file ../examples\MVC-006F_resized.jpg process took: 1.034174292926072[sec]	0.1067616000000271[sec]
file ../examples\MVC-007F_resized.jpg process took: 1.0902754104177532[sec]	0.13185090000001765[sec]
file ../examples\MVC-008F_resized.jpg process took: 1.0168617126954764[sec]	0.1242230999999947[sec]
file ../examples\MVC-009F_resized.jpg process took: 1.0469796640413715[sec]	0.12896109999996952[sec]
file ../examples\MVC-010F_resized.jpg process took: 1.0611284737877344[sec]	0.14778929999999946[sec]
file ../examples\MVC-011F._resized.jpg process took: 1.14235427631362[sec]	0.12895660000003772[sec]
file ../examples\MVC-012F._resized.jpg process took: 1.0697627703647363[sec]	0.10053719999996247[sec]
file ../examples\MVC-013F._resized.jpg process took: 1.0493666960646948[sec]	0.12954070000000684[sec]
file ../examples\MVC-014F._resized.jpg process took: 1.0402166510145996[sec]	0.12878760000000966[sec]
file ../examples\MVC-015F._resized.jpg process took: 1.0445196867826887[sec]	0.12921240000002854[sec]

Results:



References:

[1] http://www.robots.ox.ac.uk/~vgg/software/via/via_demo.html

[2] https://imgaug.readthedocs.io/en/latest/index.html

[3] https://cv-tricks.com/object-detection/faster-r-cnn-yolo-ssd/

[4] https://github.com/pierluigiferrari/ssd_keras

Project files:

.\myFiles\ runModelInfrance.py – This script would run on a given folder an it would use the trained model to find all the balls. Usage: "python runModelInfrance.py <folder with images>"

. \myFiles\augmentData.py – This script was used to augment my data.

. \myFiles\resizeImgs.py – This script was used to resize my data images.

.\myFiles\ my_pip_list.txt – If you are wondering about my environment

.\ssd512_training_transfer_learning.ipynb – jupyter notebook that I used to train and evaluate the model

.\weight_sampling_tutorial.ipynb – This jupyter notebook would subsample the a model weights, that was trained on a different number of classes then what I have to fit the number of classes I have.