Object Detection methods for wildlife species monitoring

Yulia Pavlova

project report

9 September 2019

The objective of the task was to create a program to classify wild-life animals from the photos taken by hunter cameras. The program (the first version of it) is using a TensorFlow-trained classifier to detection. The model was retrained for the specified task using images from Loimaa 2017, plus additional data of species called 'Roe deer'. I have used Single Shot Multibox Detector (SSD) neural network system.

The results of object detector model are shown below (Fig.1-Fig.4). The program draws boxes around detected animals and identifies species of the animals.

Fig. 1.

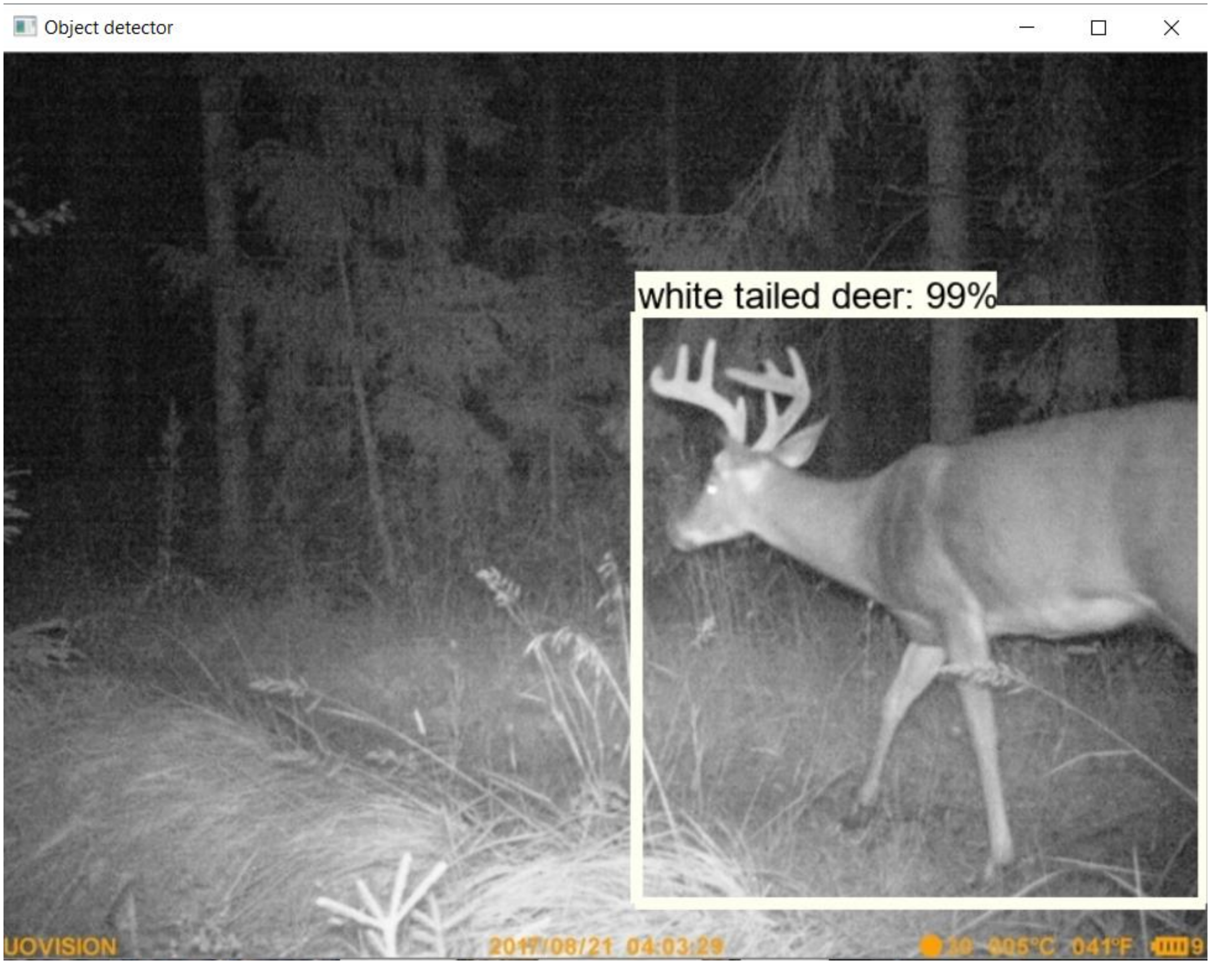


Fig.2

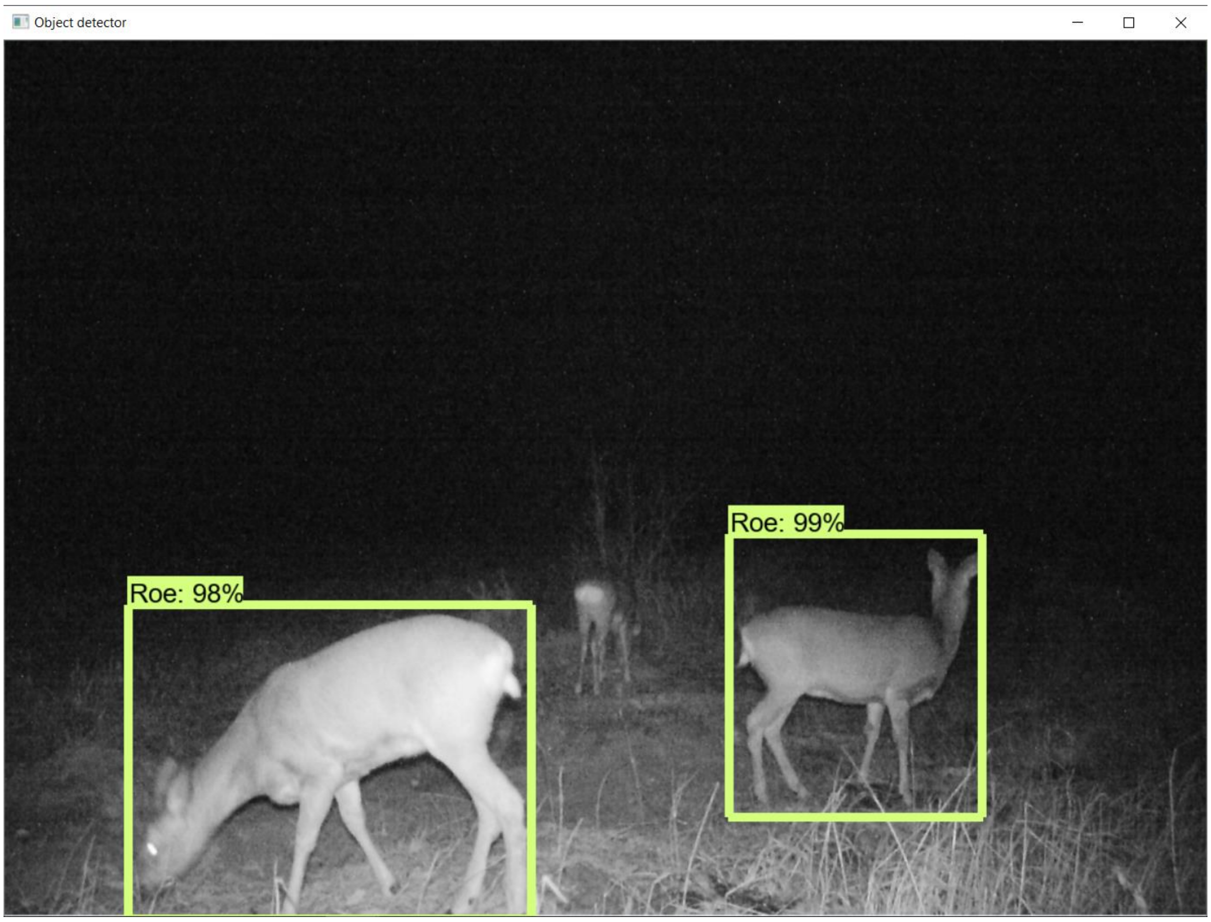


Fig.3

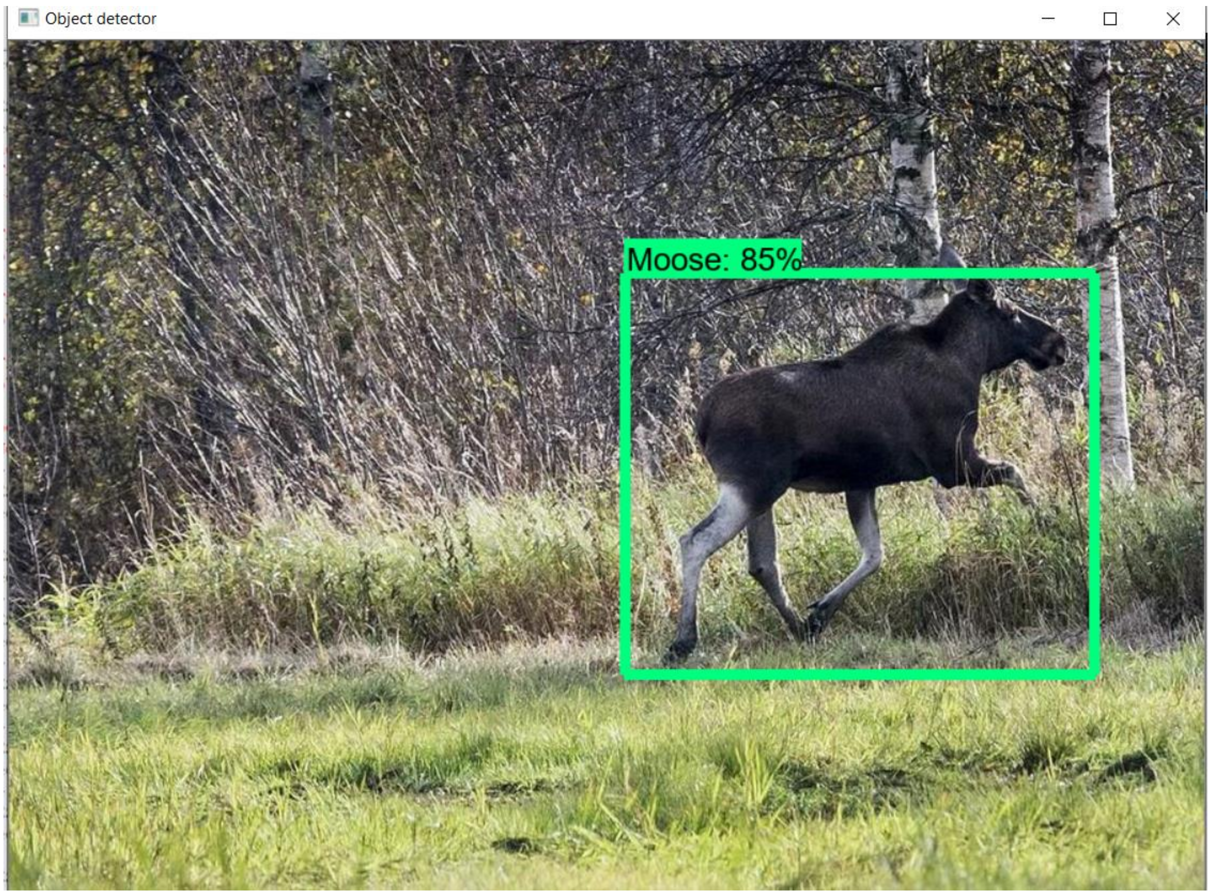
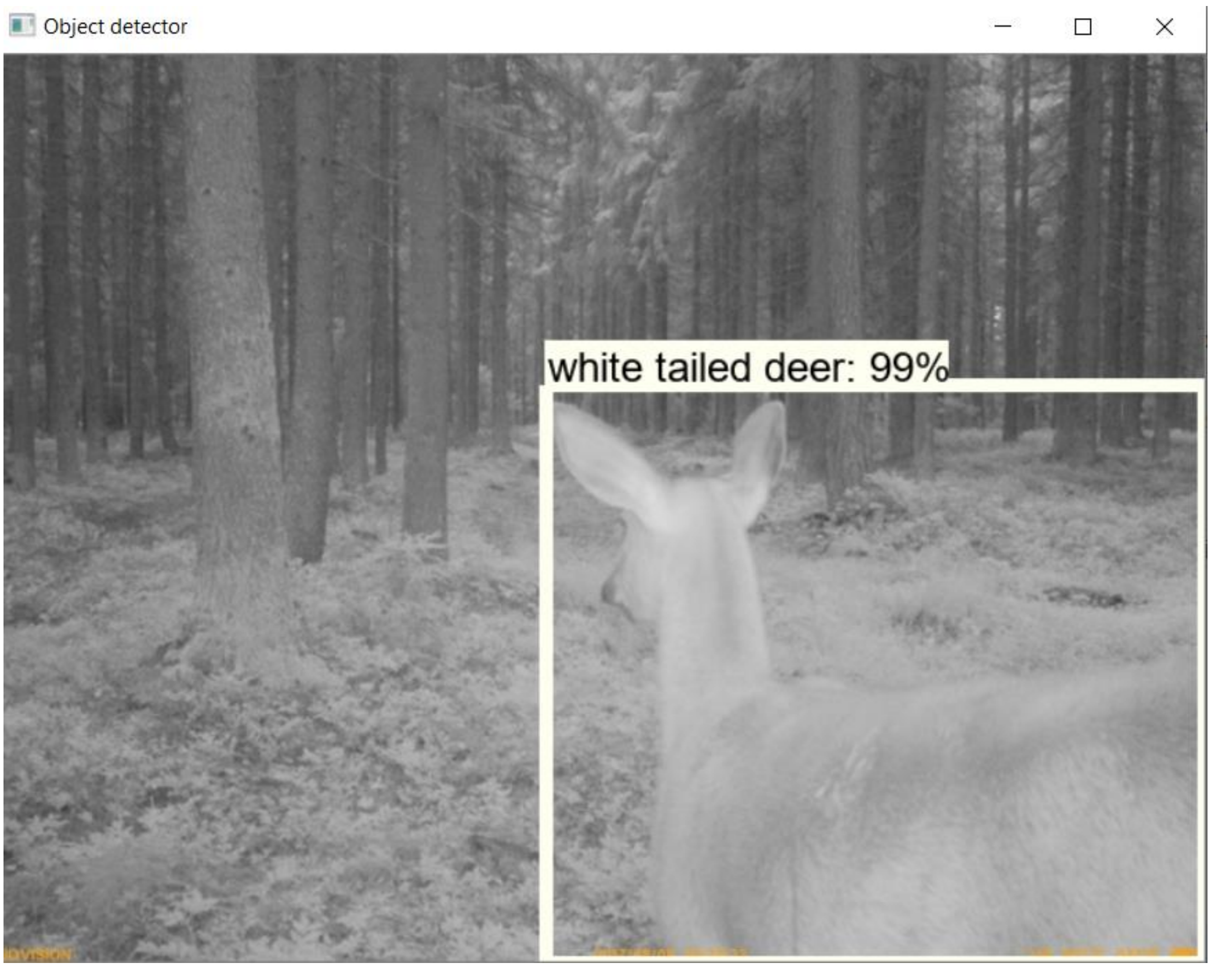


Fig.4



Initial observations show that the part model of identification the species works well, however, detecting part of model that draws boxes has some room for improvements: in many cases it fails to detect animals.

Secondly, the work on describing statistics of the model is not completed. This is some work for the future.

Thirdly, the detection can be performed on the video stream (the code is also provided).

Below I describe the steps and provide the useful links.

Installation

https://tensorflow-object-detection-api-tutorial.readthedocs.io/en/latest/install.html

An important start of the project was related to installation of TensorFlow object detection API. The link above provides all necessary steps of the process. Administrative rights are necessary. An optional but highly recommended prerequisite is Anaconda tool that helps to manage virtual environments. For this task I have chosen GPU version of TensorFlow, which is commonly considered to be faster. However, installation process is somewhat more demanding due to need of additional installation of Graphics and CUDE drivers.

When installation process is all set up, it is advisable to create a new directory – 'wild-life_animal_detection', which contains annotations, images (test/train), pre-trained-model, training, prog-utils, readme.md (contains project description). This is done for organizational purposes.

- data
- images (test/train)
- object_detection
    - training
    - inference_graph
- readme.md (contains project description).

Gather data, data preprocessing and labelling steps

Now that the TensorFlow Object Detection API is all set up and ready to go, we need to provide the images it will use to train a new detection classifier. TensorFlow needs hundreds of images of an object to train a good detection classifier. To train a robust classifier, the training images should have random objects in the image along with the desired objects and should have a variety of backgrounds and lighting conditions. There should be some images where the desired object is partially obscured, overlapped with something else, or only halfway in the picture.

Make sure the images aren't too large. They should be less than 200KB each, and their resolution shouldn't be more than 720x1280. The larger the images are, the longer it will take to train the classifier. *Resizer.py* script (in prog_utils folder) has been used to reduce the size of the images.

Annotations

With all the pictures gathered, it's time to label the desired objects in every picture. *LabelImg* is a great tool for labeling images, and its GitHub page has very clear instructions on how to install and use it . Download and install *LabelImg* https://tensorflow-object-detection-api-tutorial.readthedocs.io/en/latest/install.html#labelimg-install

Open each folder, select a photo, draw a box around each animal in each image and assign a tag. Repeat the process for all the images in the directory. LabelImg saves a .xml file containing the label data for each image. These .xml files then later are used to generate TFRecords, which are one of the inputs to the TensorFlow trainer. Once all images are labeled and saved each image, there will be one .xml file for each image.

After that, I randomly move 10% of them (images and .xml files) to the images\test directory, and 90% of them to the images\train directory. For that I use another script *move_split,py* (in prog_utils folder).

Training

With the images labeled, need to generate the TFRecords that serve as input data to the TensorFlow training model. Here I used the *xml_to_csv.py* and *generate_tfrecord.py* scripts, with some slight modifications, from

https://github.com/datitran/raccoon_dataset

First, the image .xml data will be used to create .csv files containing all the data for the train and test images. From the \object_detection folder, issue the following command in the Anaconda command prompt: python *xml_to_csv.py*.

This creates a *train_labels.csv* and *test_labels.csv* file in the \object_detection\data folder.

Next, open the generate_tfrecord.py file in a text editor. Replace the label map starting at with your own label map, where each object is assigned an ID number. This same number assignment will be used when configuring the *animal_mask.pbtxt*. In this exercise, I use four tags: rabbit, roe, moose, and white tailed deer (there is not enough images of other species from Loimaa 2017).

Then, generate the TFRecord files by issuing these commands from the \object_detection folder:

```
python generate_tfrecord.py --csv_input=images\train_labels.csv --
image_dir=images\train --output_path=train.record
python generate_tfrecord.py --csv_input=images\test_labels.csv --
image_dir=images\test --output_path=test.record
```

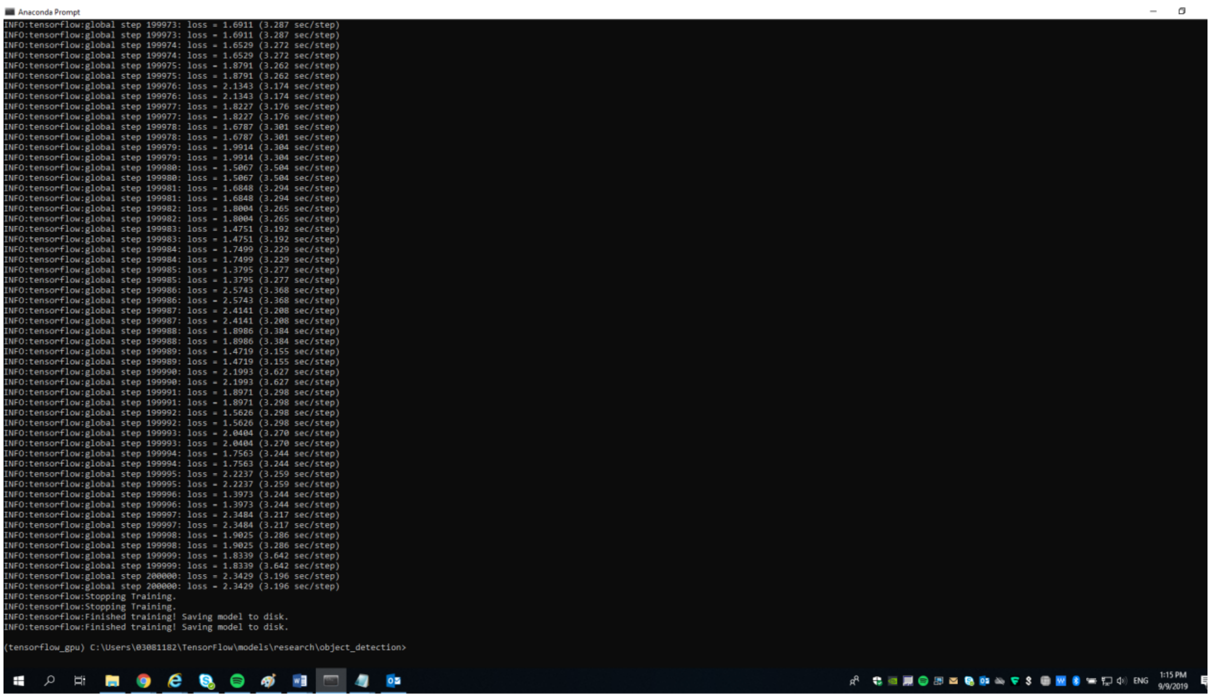
https://github.com/tensorflow/models/blob/master/research/object_detection/object_detection_tutorial.ipynb

Finally, the object detection training pipeline is configured. It defines which model and what parameters are used for training. Navigate to \object_detection\samples\configs and copied the *ssd_inception_v2_pets.config* file into the \object_detection\training directory.

After that training of the model can begin using *train.py* (in anaconda prompt from object_detection folder)

```
python train.py --logtostderr --train_dir=training/ --
pipeline_config_path=training/faster_rcnn_inception_v2_pets.config
```

Training takes several days (about 5 days), I ran it up to 200,000 steps.



```
INFO:tensorflow:global step 199973: loss = 1.6911 (3.287 sec/step)
INFO:tensorflow:global step 199973: loss = 1.6911 (3.287 sec/step)
INFO:tensorflow:global step 199974: loss = 1.6529 (3.272 sec/step)
INFO:tensorflow:global step 199974: loss = 1.6529 (3.272 sec/step)
INFO:tensorflow:global step 199975: loss = 1.8791 (3.262 sec/step)
INFO:tensorflow:global step 199975: loss = 1.8791 (3.262 sec/step)
INFO:tensorflow:global step 199976: loss = 2.1343 (3.174 sec/step)
INFO:tensorflow:global step 199976: loss = 2.1343 (3.174 sec/step)
INFO:tensorflow:global step 199977: loss = 1.8227 (3.176 sec/step)
INFO:tensorflow:global step 199977: loss = 1.8227 (3.176 sec/step)
INFO:tensorflow:global step 199978: loss = 1.6787 (3.301 sec/step)
INFO:tensorflow:global step 199978: loss = 1.6787 (3.301 sec/step)
INFO:tensorflow:global step 199979: loss = 1.9914 (3.304 sec/step)
INFO:tensorflow:global step 199979: loss = 1.9914 (3.304 sec/step)
INFO:tensorflow:global step 199980: loss = 1.5067 (3.504 sec/step)
INFO:tensorflow:global step 199980: loss = 1.5067 (3.504 sec/step)
INFO:tensorflow:global step 199981: loss = 1.6848 (3.294 sec/step)
INFO:tensorflow:global step 199981: loss = 1.6848 (3.294 sec/step)
INFO:tensorflow:global step 199982: loss = 1.8004 (3.265 sec/step)
INFO:tensorflow:global step 199982: loss = 1.8004 (3.265 sec/step)
INFO:tensorflow:global step 199983: loss = 1.4751 (3.192 sec/step)
INFO:tensorflow:global step 199983: loss = 1.4751 (3.192 sec/step)
INFO:tensorflow:global step 199984: loss = 1.7499 (3.229 sec/step)
INFO:tensorflow:global step 199984: loss = 1.7499 (3.229 sec/step)
INFO:tensorflow:global step 199985: loss = 1.3795 (3.277 sec/step)
INFO:tensorflow:global step 199985: loss = 1.3795 (3.277 sec/step)
INFO:tensorflow:global step 199986: loss = 2.5743 (3.368 sec/step)
INFO:tensorflow:global step 199986: loss = 2.5743 (3.368 sec/step)
INFO:tensorflow:global step 199987: loss = 2.4141 (3.208 sec/step)
INFO:tensorflow:global step 199987: loss = 2.4141 (3.208 sec/step)
INFO:tensorflow:global step 199988: loss = 1.8986 (3.384 sec/step)
INFO:tensorflow:global step 199988: loss = 1.8986 (3.384 sec/step)
INFO:tensorflow:global step 199989: loss = 1.4719 (3.155 sec/step)
INFO:tensorflow:global step 199989: loss = 1.4719 (3.155 sec/step)
INFO:tensorflow:global step 199990: loss = 2.1993 (3.627 sec/step)
INFO:tensorflow:global step 199990: loss = 2.1993 (3.627 sec/step)
INFO:tensorflow:global step 199991: loss = 1.8971 (3.298 sec/step)
INFO:tensorflow:global step 199991: loss = 1.8971 (3.298 sec/step)
INFO:tensorflow:global step 199992: loss = 1.5626 (3.298 sec/step)
INFO:tensorflow:global step 199992: loss = 1.5626 (3.298 sec/step)
INFO:tensorflow:global step 199993: loss = 2.0404 (3.270 sec/step)
INFO:tensorflow:global step 199993: loss = 2.0404 (3.270 sec/step)
INFO:tensorflow:global step 199994: loss = 1.7563 (3.244 sec/step)
INFO:tensorflow:global step 199994: loss = 1.7563 (3.244 sec/step)
INFO:tensorflow:global step 199995: loss = 2.2237 (3.259 sec/step)
INFO:tensorflow:global step 199995: loss = 2.2237 (3.259 sec/step)
INFO:tensorflow:global step 199996: loss = 1.3973 (3.244 sec/step)
INFO:tensorflow:global step 199996: loss = 1.3973 (3.244 sec/step)
INFO:tensorflow:global step 199997: loss = 2.3484 (3.217 sec/step)
INFO:tensorflow:global step 199997: loss = 2.3484 (3.217 sec/step)
INFO:tensorflow:global step 199998: loss = 1.9025 (3.286 sec/step)
INFO:tensorflow:global step 199998: loss = 1.9025 (3.286 sec/step)
INFO:tensorflow:global step 199999: loss = 1.8339 (3.642 sec/step)
INFO:tensorflow:global step 199999: loss = 1.8339 (3.642 sec/step)
INFO:tensorflow:global step 200000: loss = 2.3429 (3.196 sec/step)
INFO:tensorflow:global step 200000: loss = 2.3429 (3.196 sec/step)
INFO:tensorflow:Stopping Training.
INFO:tensorflow:Stopping Training.
INFO:tensorflow:Finished training! Saving model to disk.
INFO:tensorflow:Finished training! Saving model to disk.

(tensorflow_gpu) C:\Users\03081182\TensorFlow\models\research\object_detection>
```

Each step of training reports the loss. It will start high and get lower and lower as training progresses. The loss numbers will be different if a different model is used. MobileNet-SSD starts with a loss of about 20, and should be trained until the loss is consistently under 2. We go loss around 2, so some improvement to the dataset is necessary, and also fine-tuning of the model. More recommendations can drawn upon analyzing model's statics.

The progress of the training job can also been viewed by using TensorBoard. To do this, open a new instance of Anaconda Prompt, activate the tensorflow virtual environment, and from object_detection directory, and issue the following command:

```
tensorboard --logdir=training
```

This will create a webpage on your local machine at YourPCName:6006, which can be viewed through a web browser. The TensorBoard page provides information and graphs that show how the training is progressing. One important graph is the Loss graph, which shows the overall loss of the classifier over time (see it inattachment).

TensorBoard

In attachment you can see statistics from the tensorboard. It shows that model converges nicely, even without tuning and altering hyperparameters

https://www.tensorflow.org/guide/summaries_and_tensorboard

Testing the model

Now that training is complete, the last step is to generate the frozen inference graph (.pb file). From the \object_detection folder, issue the following command:

```
python export_inference_graph.py --input_type image_tensor --pipeline_config_path training/ssd_inception_v2_pets.config --trained_checkpoint_prefix training/model.ckpt-200000 --output_directory inference_graph
```

This creates a frozen_inference_graph.pb file in the \object_detection\inference_graph folder. The .pb file contains the object detection classifier.

To test your object detector, move a picture of the object or objects into the \object_detection folder, and change the IMAGE_NAME variable in the *Object_detection_image.py* to match the file name of the picture. Alternatively, you can use a video of the objects (using *Object_detection_video.py*), or just plug in a USB webcam and point it at the objects (using *Object_detection_webcam.py*).

To run any of the scripts, type “idle” in the Anaconda Command Prompt (with the “tensorflow” virtual environment activated) and press ENTER. This will open IDLE, and from there, you can open any of the scripts and run them. The object detector will initialize for about 10 seconds and then display a window showing any objects it’s detected in the image. Results of the object_detector_image.py are presented in the beginning of this document Fig. [1—4].

Other information about modern neural networks

Modern methods in object detection

Object detection is a common term for computer vision techniques classifying and locating objects in an image. Modern object detection is largely based on use of convolutional neural networks. Some of the most relevant system types today are Faster R-CNN, R-FCN, Multibox Single Shot Detector (SSD) and YOLO (You Only Look Once).

Original **R-CNN** method worked by running a neural net classifier on samples cropped from images using externally computed box proposals (=samples cropped with externally computed box proposals; feature extraction done on all the cropped samples). This approach was computationally expensive due to many crops; Fast RCNN reduced the computation by doing the feature extraction only once to the whole image and using cropping on the lower layer (=feature extraction only once on the whole image; samples cropped with externally computed box proposals). Faster RCNN goes a step further and used the extracted features to create class-agnostic boxproposals (=feature extraction only once on the whole image; no externally computed box proposals). R-FCN is like Faster R-CNN but the feature cropping is done in a different layer for increased efficiency.

YOLO (You Only Look Once) works on different principle than the aforementioned models: it runs a single convolutional network on the whole input image (once) to predict bounding boxes with confidence scores for each class simultaneously. The advantage of the simplicity of the approach is that the YOLO model is fast (compared to Faster R-CNN and SSD) and it learns a general representation of the objects. This increases localization error rate (also, YOLO does poorly with images with new aspect ratios or small object flocked together) but reduces false positive rate.

Single Shot Multibox Detector (SSD) differs from the R-CNN based approaches by not requiring a second stage per-proposal classification operation. This makes it fast enough for real-time detection applications. However, this comes with a price of reduced precision. Note that "**SSD with MobileNet**" refers to a model where model meta architecture is SSD and the feature extractor type is MobileNet.

Speed-accuracy tradeoff

Many modern object detection applications require real-time speed. Methods such as YOLO or SSD work that fast, but this tends to come with a decrease in accuracy of predictions, whereas models such as Faster R-CNN achieve high accuracy but are more expensive to run. The cost in model speed depends on the application: With larger images (e.g. 600x600) SSD works comparable to more computationally expensive models such Faster R-CNN, even as on smaller images its performance is considerably lower.

TensorFlow Object Detection API with SSD model with Mobilenet

TensorFlow Object Detection API is “an open source framework built on top of TensorFlow” that aims to make it easy to “construct, train and deploy object detection models”. To achieve this, TensorFlow Object Detection API provides the user with multiple pre-trained object detection models with instructions and example codes for fine-tuning and using the models for object detection tasks. TensorFlow Object Detection API can be used with different pre-trained models. In this work, two models were chosen and tested. A SSD model with Mobilenet (ssd_mobilenet_v1_coco) was the second choice. The original model had been trained using MSCOCO Dataset which consists of 2.5 million labeled instances in 328 000 images, containing 91 object types such as “person”, “cat” or “dog”. The ssd_mobilenet_v1_coco-model is reported to have mean Average Precision (mAP) of 21 on COCO dataset. In this work, the SSD model was tested both as pre-trained without fine tuning and as fine-tuned for our own dataset.1

TensorBoard SCALARS GRAPHS DISTRIBUT INACTIVE

☐ Show data download links

☐ Ignore outliers in chart scaling

Tooltip sorting method: default

Smoothing

0.6

Horizontal Axis

STEP RELATIVE

WALL

Runs

Write a regex to filter runs

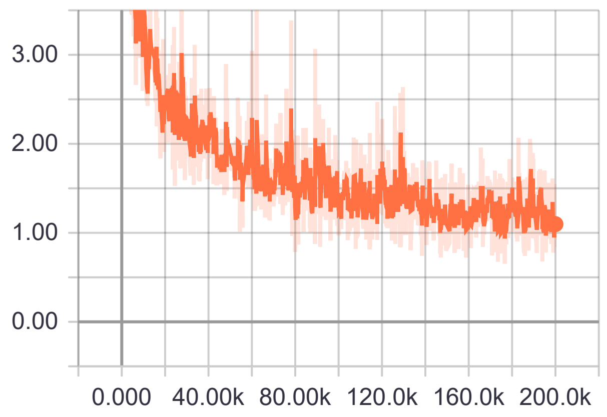
☐ ○ .

Filter tags (regular expressions supported)

LearningRate	1
Loss	2
Losses	5

Loss/classification_loss

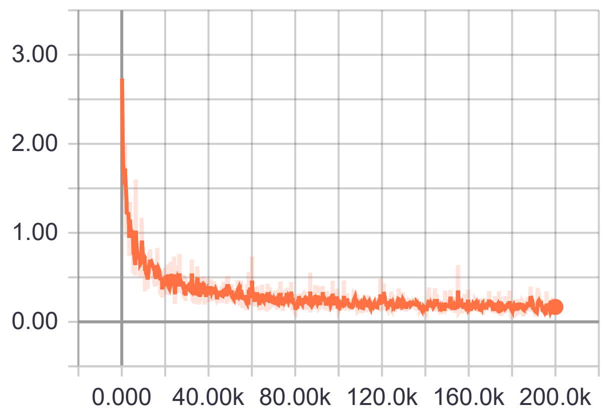
tag: Losses/Loss/classification_loss



run to download CSV JSON

Loss/localization_loss

tag: Losses/Loss/localization_loss



run to download CSV JSON

TotalLoss

tag: Losses/TotalLoss