Signate 3rd AI edge competition
Solution of IRAFM-AI team, 2nd place

Petr Hurtik and Marek Vajgl, IRAFM, University of Ostrava

July 5, 2020

1 General information

This section describes the overview of the boxes detection process, hardware specifications used for training, and also some additional notes related to the competition solution. All source codes are available online[1]. Note that the repository may continue updated description and additional information for better code understanding – see `readme.md` file at the repository.

The generic pipeline of boxes detection and classification is as follows:

1. Detect bounding boxes of potential objects by multiple Poly-YOLOs neural networks.
2. Aggregate area-matching bounding boxes by weighted box fusion.
3. Refine each bounding box by a scheme based on EfficientNet
4. Classify every bounding box into the one of three categories car/pedestrian/background
5. Match IDs of previous boxes and current boxes via siamese network and our solver, assign new IDs for new boxes
6. Do post-processing: Delete boxes with the number of occurrences less than 3 and fill boxes missing in the current frame, but existing in preceding and succeeding frame

For training, we have used two GPUs/computers. One has RTX2080, and the second has RTX2060. From the implementation point of view, we have used Python 3.7 with Keras 2.3 and TensorFlow 1.15. The inference was realized using Signate's docker image with TensorFlow 2.0. Another used libraries were OpenCV and NumPy.

For the implementation, partially IntelliJ PyCharm IDE and Jupyter Notebooks were used. **The below mentioned training information and description refers to submitted Jupyter Notebooks (*.ipynb) and python files (*.py)** files in which the required functionality is.

Generally, we observed that we utilize GPU during predictions for only approximately 45 %. A possible reason is the Python back-end, which is not implemented precisely and improvement of this source code may lead to a significant reduction of calculation time.

Lessons learned (what we tried and did not work):

- the usage of soft-NMS instead of hard-NMS,
- pretraining on Cityscapes or Waymo datasets,
- the usage of trackers from OpenCV,
- the usage only top-part of pedestrian boxes for tracking.

Interesting hacks:

- Remove out boxes with a high aspect ratio (pedestrians> 4.2, cars> 2.1)
- Crop images 100px from top and bottom. That is a dead area; there is nothing to detect. Such images help us to train/predict faster.
- Refine boxes location using specific NN for that purpose.
- Write a task-specific tracker/matcher to track the object IDs.

[1] https://gitlab.com/irafm-ai/signate_3rd_ai_edge_competition

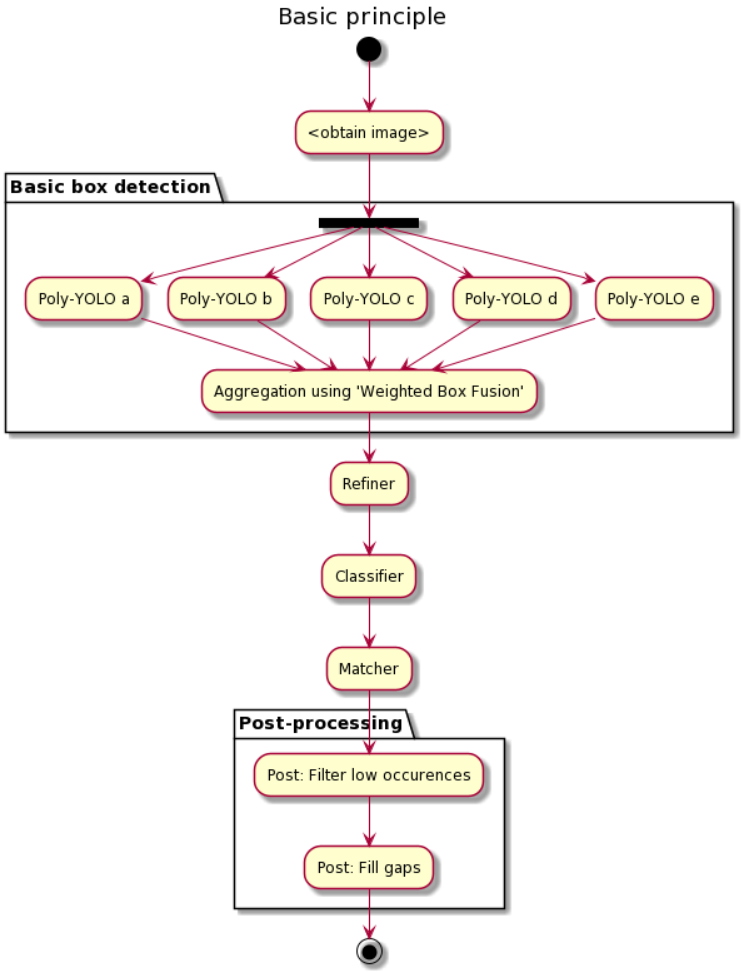

Figure 1: Basic pipeline

2 Poly-YOLO

Poly-YOLO is our development [1] of YOLOv3, where we significantly increase accuracy by solving YOLO's principal issues. It can also realize instance segmentation, but it is not the aim of this competition, so we haven't used this part of the implementation. For more technical details, see our repository[2]. In our solution, we use the Poly-YOLO network based on the repository improved by the usage of Atrous Pooling Pyramid blocks to increase the receptive field. We use five such YOLOs trained for various resolutions (below notated has height × width in pixels) and batch sizes (limitations of our hardware set batch sizes); the file-name refers to the file used for model learning:

1) 448×864, batch size 2, file `yolo_v4_wo_poly_multiscale.py`
2) 352×704, batch size 3, file `yolo_v4_wo_poly_multiscale.py`
3) 224×448, batch size 10, file `yolo_v4_wo_poly_multiscale_v4.py`
4) 960×1952, batch size 1, trained only for pedestrians only, file `yolo_v4_full_res.py`
5) 544×1120, batch size 1, file `yolo_v4_wo_poly_multiscale_v4.py`

In addition, Poly-YOLOs marked as 3 and 5 use mish activations (instead of leaky ReLu), DIOU loss (instead of separate x-y and width-height loss), and label smoothing.

The Poly-YOLO marked as 4 is a special one because it has a high resolution, so it has a reduced number of parameters. Furthermore, we have trained it on non-augmented data. The rest of the models are trained for augmented data.

The process of training contains two phases. In the first phase, every Poly-YOLO network should be trained from scratch for 30-50 epochs with the functionality of reducing the learning rate on the plateau.

[2]https://gitlab.com/irafm-ai/poly-yolo

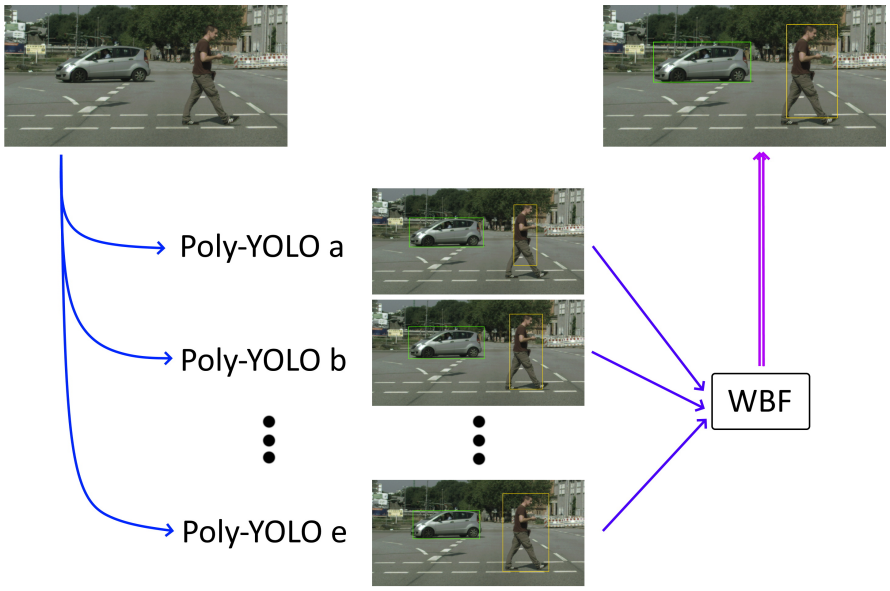

Figure 2: Poly-YOLOs boxes detection. The image is placed as input for every of the five YOLO networks. Every network suggests its own boxes estimation; those results are aggregated using 'Weighted Box Fusion,' and the final estimation of the first step is produced.

Then, in the second phase, the best model according to valid loss is taken, loaded, the learning rate is halved, and the training is performed again for additional 30-50 epochs. We have stopped the learning process once a model starts to overfit for several epochs. The second phase is repeated, so each model is trained three times. The only difference is for model 4, which uses the cyclic learning rate instead of reducing LR on the plateau and was trained only twice.

For training Poly-YOLO, the following scripts are necessary:

- *prepare_data_decode_movies*: takes the movies and decode into frames. At the same time, it crops every image by cutting the top and bottom part of the image such that for height $y \in [100, ..., 1050]$.
- *prepare_data_prepare_labels*: takes the jsons and converts them into txt format of labels that is used for YOLO training. It extracts only classes Car and Pedestrian (or only Pedestrians for YOLO marked as 4 – see used YOLO listing above).
- *inpaint_data*: creates new images using the following scheme: take an input image and put into it random boxes from other images. This creates "new", inpainted images for the training.
- *mosaic_data* creates new image samples where each created image sample consists of two random images split in half vertically.

Again note that the implementations are provided in the format of Jupyter Notebooks and python files. All the scripts mentioned here (and later) are available from the folder `source_codes` from the GitLab repository mentioned in footnote[1].

For that, the training process is as follows:

1. Decode movies using `prepare_data_decode_movies`.
2. Prepare labels using `prepare_data_prepare_labels` as follows:
   (a) Execute the script `prepare_data_prepare_labels`. This script creates the whole *training txt file*, typically called as `data_for_yolo_training.txt`.
   (b) From this *training txt file*, manually select data for validation (we have selected all the lines for the "movie_01") and cut them out (keep them for the next in the next step).
   (c) Create a new file, *validating txt file* and put the data obtained from the previous step into this file.
3. Prepare additional data using scripts `inpaint_data` and `mosaic_data`.
   (a) Execute the script `inpaint_data`, which will create a new output file.
   (b) Take all the content of the created output file from the previous and append it at the end of

the aforementioned *training txt file*.
   (c) Execute the script `mosaic_data`, which will create another new output file.
   (d) Again, take all the content of the created output file paste it at the end of the aforementioned *training txt file*.
4. Set paths to the *training text file* and *validating text file* inside a corresponding YOLO python file. Set phase to 1. Run the .py file (see the relative .py files in the Poly-YOLO model listing above). Set resolution and batch size according to the description above.
5. After training, take model (`.h5` file) with the lowest validation loss, set path to in inside the `.py` file, change phase to 2, decrease learning rate, and run again.
6. Additionally, repeat the previous step.

For the inference:
For the inference, we load the models, set they resolution as it is defined above, and loaded the same files defining anchors and classes as were used for training. We realized the predictions, concatenated them, sort according to confidence in ascending order, and merge boxes with IOU bigger than 0.45. The aggregation is realized with weights 7:3, where the box with lower confidence has a lower weight.

3 Refiner

For the classification process, we needed to improve (for some cases significantly) the precision of the estimated box location.

Therefore, we have trained a network (effnetb3 backbone) – called *the refiner* – that takes a crop of an image according to the detected box (with slight padding) and predicts precise box (see Figure 3). Because it always takes an image with a single object, it is a trivial task. We have trained the network for DIOU loss. The prediction of all detected boxes by YOLO is made in a single batch, so it is really fast. The refiner increases the boxes' precision significantly.

As mentioned before, for the refiner, every box needs a little pre-processing:

1. before cropping, the box boundary is extended by small padding so the box boundary can be extended by the refiner's prediction), then the crop for the refiner is obtained,
2. every crop is resized to match the refiner's input square window.

Obviously, the refiner's result must be recalculated to match the coordinates of the original image.

To **train the refiner** use firstly script `prepare_data_refiner`, which will create crops from the training data. The script has to be used twice, separately for training and validation data. Then, run script *train_refiner* that will train the refiner. The process is similar to the training of YOLO: train it, take the best model according to the validation score, load it, and train for the second time.

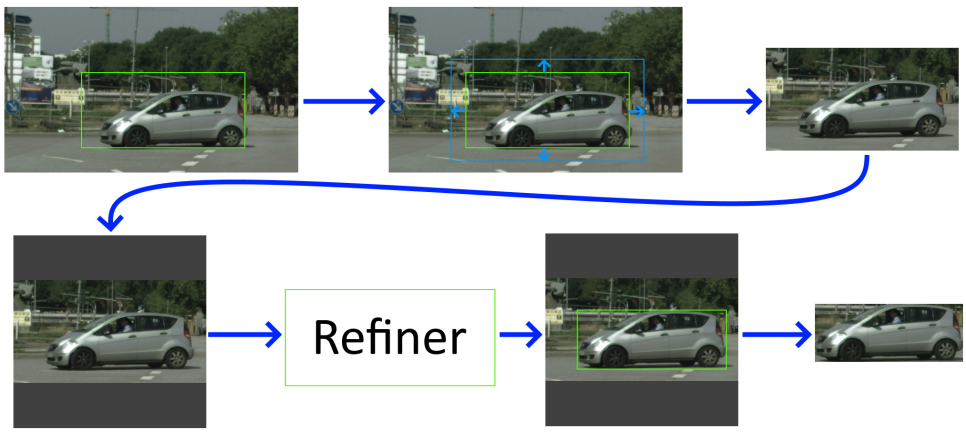

Figure 3: Refiner processing scheme. 1) The estimated box boundary is 2) extended by small padding. Then 3) crop of the image is obtained, and 4) matched into the refiner input scale. Refiner 5) updates the boundaries to produce updated (more precise) 6) box boundaries.

4 Classifier

Because of an imbalance between car/pedestrian classes, we trained the EffnetB3 network, which classifies the boxes, trained with focal loss – *the classifier*. The network also filters out false positives, which further increases the score.

The classifier is distinguishing three classes: a car, a pedestrian, and a background. The last one is used as a further filter to detect unwanted boxes. Boxes marked as background are expected to be false positives and omitted from further processing.

The data for classifier are prepared by script `prepare_data_classifier` which generate the cropped images. It needs a file including all boxes for all classes. It can be generated by script `prepare_data_prepare_labels`.

The classifier is trained using script `train_classifier` in two stages. Firstly, we trained the network and made predictions for the training set. By visualization of the wrong predictions, we have found that the train set contains approximately 300 incorrect labels. Therefore, we have trained the network again without such labels. This procedure increases the accuracy of up to 99.8% of correctly detected classes. Like the refiner, the classifier realizes all boxes' predictions in one step with a big batch size.

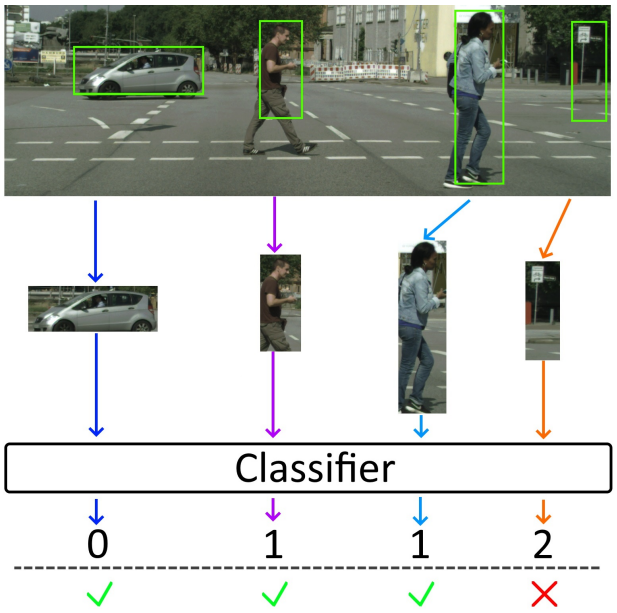

Figure 4: Classifier

5 Matcher

Another requirement was to track objects between the frames during the video. Every object is identified by its ID.

In the beginning, we have used already known algorithms for tracking the object during the video frames, called trackers. We have used publicly accessible and implemented tracking algorithm from Python opencv-contrib library, but the results were unsatisfactory.

Therefore, we have implemented our custom approach used to *match* objects between frames, therefore called *the matcher*.

The matcher is based on a siamese neural network with contrastive loss and effnetb0 backbone, calculating similarities between the boxes. According to this similarity, we are able to map objects between the frames and assign the correct IDs to them (see Figure 5).

The data for the matcher is prepared using script `prepare_data_matcher`. Training is realized by `train_matcher`. In contrast to the refiner and classifier, we observed that the matcher tends to overfit for B3 efficientnet. Therefore we selected the version with fewer parameters, namely B0. As same as the

other models, the matcher is trained in two stages: train, select the best model according to the valid loss, load the model, reduce learning rate to the half and train for the second time.

As post-processing of the matcher, we do match the similar boxes not only between the adjacent frames but also looking into the history for the previously found objects not detected in the preceeding frame.

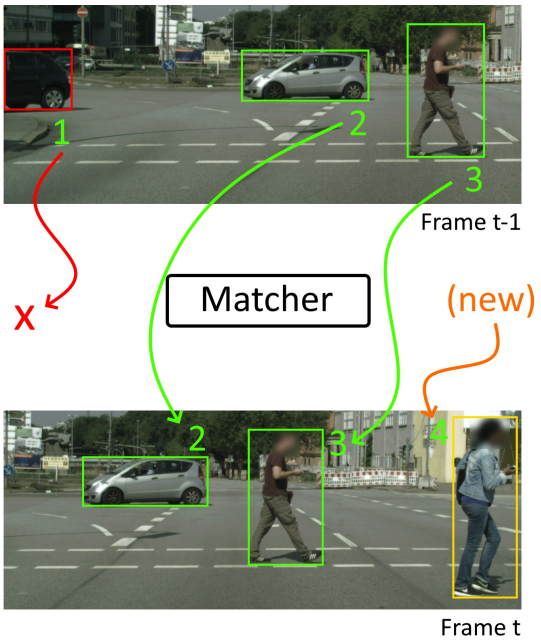

Figure 5: Matcher. Matcher calculates the box similarities, using which the boxes between the frames are assigned together.

6 Postprocessing

After the whole process, we have suggested several small improvements to increase the final quality. The major used were the deletion of small boxes, deletion of boxes with two or fewer occurrences (both required by competition rules), and adding boxes missing between frames.

6.1 Deletion of small boxes

After predictions of YOLO, we filter the detected boxes to boxes with the size of at least 900 pixels. In the end (after refiner), we have filtered out boxes that are smaller than 1024 pixels.

6.2 Deletion of boxes with low occurrence

We walk through all the frames, and for every object (that is, for every ID) we look for the number of adjoining occurrences. If this number is lower than 3, we delete boxes with such ID from the result.

6.3 Filling gap of missing object occurences in adjacent frames

Our modified matcher/solver process is able to track object ID not only between adjoining frames but also deeper in history. There may occur a situation when an object (that is, an ID) is detected in frame at time $t-2$ and at time t, but such box is missing in the frame at time $t-1$. For this case, we have inserted the missing box in the result of the frame $t-1$.

For this, we have experimented with this process and found that the optimal value is to fill up boxes missing in at most two adjacent frames - e.g., we will insert box with id appearing in the frames at time $t-3$ and t into the frames at time $t-2$ and $t-1$. For the box location in the missing frames, the simple linear estimation is used.

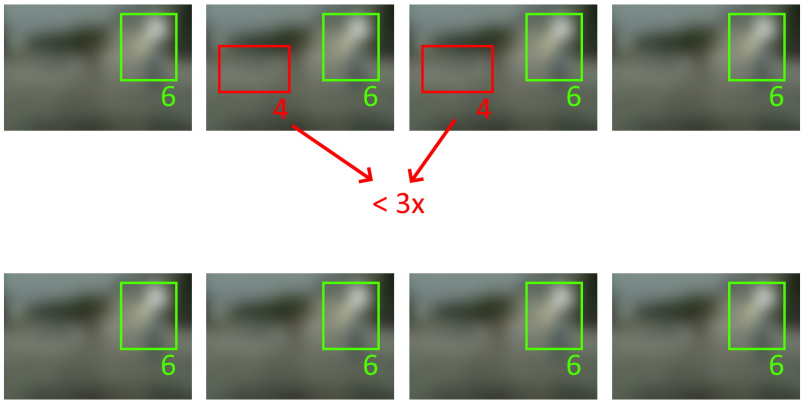

Figure 6: Occurrence based drop-out. For every ID, the number of occurrences is calculated. If the number of occurrences is lower than 3, the object is omitted from the result.

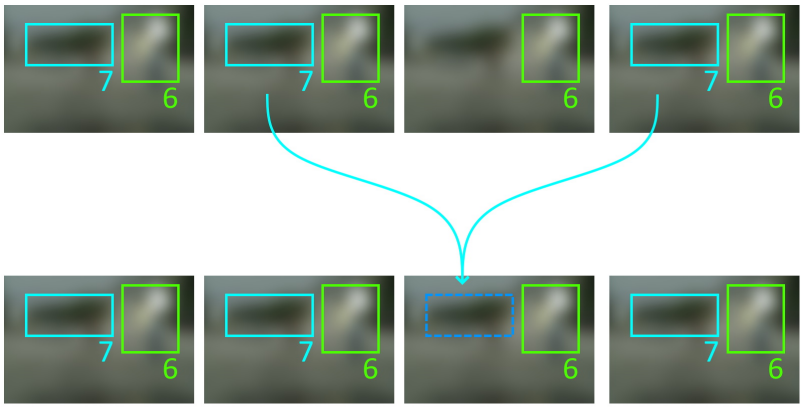

Figure 7: We are filling gaps in object occurrences in adjacent frames (top row: before, bottom row: after). In the top row, in the third image, the ID 7 is missing but is present in the neighbor frames. Therefore, this box is inserted into the third frame (bottom row).

7 Optimize models for speed

Because the saved models use variables needed only for training and the architectures use an inefficient combination of convolution + batch normalization, which can be used during inference as depth-wise convolution, we use script *optimize_models* for optimizing the models. The script takes as input the original model, realizes the optimization and produces a model that can be used only for predictions, but is much faster.

References

[1] Petr Hurtik, Vojtech Molek, Jan Hula, Marek Vajgl, Pavel Vlasanek, and Tomas Nejezchleba. Poly-yolo: higher speed, more precise detection and instance segmentation for yolov3. *arXiv preprint arXiv:2005.13243*, 2020.