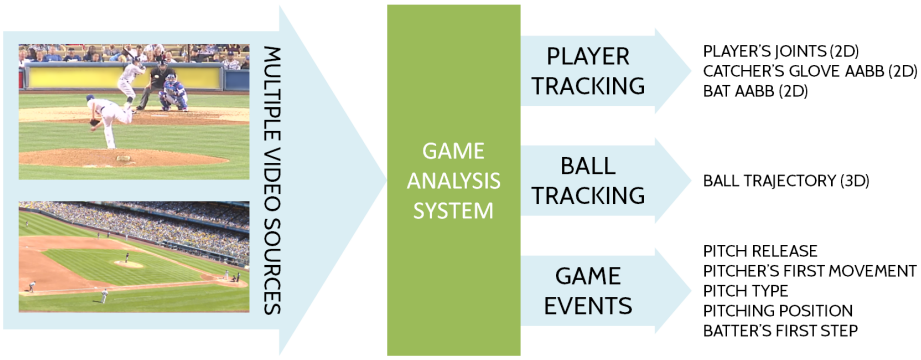
Computer Vision Methods for Baseball Analysis - Documentation

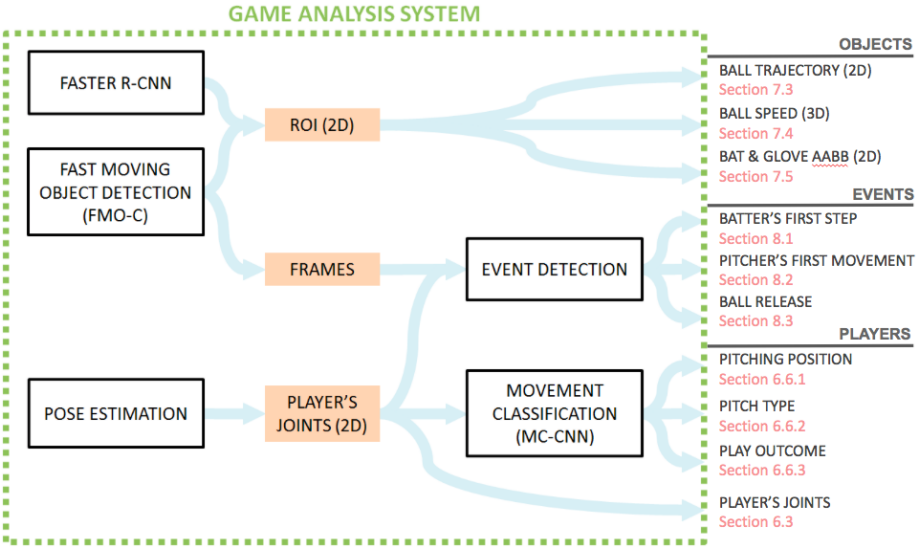
Nina Wiedemann

19.06.2018

Contents



{#fig: system_overview}



{#fig: game_analysis_system}

1 Overview (Main directory)

The proposed framework consists of three main modules: player tracking, object tracking and event detection. Different methods are combined for each module. The code is also sorted into these three modules, where player tracking is further divided into pose estimation and movement classification.

Firstly I will explain the code that is used for multiple tasks (and is thus stored in the main directory). To run experiments for a specific section of the thesis though, refer to the corresponding section in this documentation.

1.1 Dependencies

To create a temporary environment in anaconda:

```
conda env create -f environment.yml
```

Otherwise see requirements.txt file - can be installed in anaconda with

```
conda install --yes --file requirements.txt
```

or

```
while read requirement; do conda install --yes requirement; done < requirements.txt
```

Then activate the environment (and deactivate when finished):

On OSX:

```
source activate baseball_analysis
source deactivate
```

On Windows:

```
activate baseball_analysis
deactivate
```

1.1.1 Operating systems

The OpenCV version which is used here does not work on Windows properly. The videos can not be read in, probably due to an error related to ffmpeg. If this error occurs, you can use another OpenCV version: To update OpenCV, activate the environment and then type

```
conda install opencv
```

However, in the updated OpenCV version, the videos are sometimes processed differently. Thus, the notebook *4_Object_tracking/ball_speed.ipynb* might not work correctly, and when plotting on videos, the video might lag behind.

On OS X, everything should work with OpenCV 3.1.0.

1.2 Demo notebook

For a quick demonstration of each method for each task (movement classification and event detection), see the notebook *demo.ipynb*. The outputs for example videos saved in the folder *demo_data* are visualized in this notebook.

1.3 Train data

All data required for training is contained in the *train_data* folder:

1.3.1 Video data

- The major video database consists of center-field and side-view videos, one for each play (6 seconds length). The videos are sorted by data, and are stored in *train_data/ATL*.
- For the batter's first step, a separate folder of videos was created that only contains videos (and the belonging joint trajectories) in which the batter starts to run. This folder is *train_data/batter_runs/videos*
- For bat detection, high quality videos from YouTube are used. They are stored in *train_data/high_quality_videos*.
- For ball speed I use shorter videos from a public database, that show only the ball trajectory (<1 second). There are two side-view and one center-field camera for each play (folders of 6-digit-play id). They are stored in *train_data/short_ball_speed_videos*.

1.3.2 Pose estimation output

- Joint trajectories for pitcher and batter in center-field and side-view videos are saved as csv files, one for each view (cf for center-field, sv for side-view) and each player (pitcher and batter): *cf_pitcher.csv, cf_batter.csv, sv_pitcher.csv and sv_batter.csv*
- Joint trajectories for high quality videos can be found in the folder *train_data/batter_hq_joints*.

1.3.3 Metadata

- For player localization, the start position of the target player must be provided.
  - For ATL videos, *.dat* files with bounding boxes for each video are stored in the same folder as the videos.
  - For high quality videos, I manually created a file *center_dics.json* that is stored in the *train_data* folder as well.
- In order to estimate the ball release frame, the ball speed is estimated as well. To compare it to ground truth speed from Statcast, these Statcast values are saved in a separate json file (containing the speed for each side-view video): *train_data/speed_labels_sv.json*
- Manually labeled (with gradient approach) data for the batters first step: *train_data/batter_runs/labels_first_batter_test* and *train_data/batter_runs/labels_first_batter_train*.
- When a model is trained to find the batter's first step, the training data can be saved and visualized. An example is saved in *train_data/batter_first_data* and *train_data/batter_first_label*

1.4 FMO-detection

Fast Moving Object (FMO) Detection finds moving objects by thresholding difference images and searching for connected components. Here, it is used to track objects and to find the pitcher's first movement. The algorithm is build on the work in "The World of Fast Moving Objects"(Rozumnyi et al. 2017).

As FMO-detection is used for both object tracking and event detection, the script and relevant functions are stored in the main folder, in *fmo_detection.py*. The script takes a video as input, and outputs the pitcher's first movement frame index (if joint trajectories were available), ball trajectory and the motion candidates for each frame. Hyperparameters can be changed in the *config_fmo.py* file.

1.5 Neural Network related files

1.5.1 Training

For both movement classification and for event detection, ANNs are trained. Since the general code for training an ANN is the same, scripts are stored in the main folder. The *model.py* file contains different ANN models, from LSTMs to one-dimensional CNNs, while the run-files are used to start training.

- *run_thread.py* is used for classification tasks, where classes are represented as one-hot-vectors
- *run_10fold.py* is used for ten fold cross validation in the experiments
- *run_events.py* is used to train a network to find the frame index of an event, such as the moment of the batter's first step

Runner classes can be executed as Threads.

1.5.2 Testing

In the *test.py* file, any pre-trained model can be restored and data (saved as a numpy array) can be loaded to yield the corresponding labels. If labels are available, they can also be passed as an argument and the accuracy is displayed. The function is mainly used in the the different modules (movement classification and event detection), but can also be executed directly with:

test.py [-h] data_path model_path [-labels]

Arguments:

- data_path: Path to a numpy array of the data (e.g. joint trajectories)
- model_path: Path to a pre-trained model that can be restored
- -labels: Path to a numpy array with same length as the data, with labels for each data point

1.6 Utils

1.6.1 Utils for data processing

In *utils.py* functions for all kind of tasks can be found, for example calculating the accuracy per class, getting data from the csv files, shifting joint trajectories by a range of frames, etc. The functions are saved in a class *Tools*, so it is clearly visible if a function is imported from *utils*.

1.6.2 Other utils of previous versions or used for filtering

Helper files, for example for different filtering methods, are saved in the folder *utils_filtering*.

Conversions between csv to json files, saving videos as jumpy arrays, converting numpy arrays into a json file and similar helper functions can be found as well. In the experiments, only filtering is used though.

2 Pose estimation

In this folder, files for pose estimation, player localization and filtering of the joint trajectories are provided. For pose estimation itself, a pre-trained model from the work of (Cao et al. 2017) is used. All files in this directory that are not explained in this documentation belong to the code of (Cao et al. 2017).

2.1 Script for pose estimation

To test pose estimation (not my own code), use the script *pose_estimation_script.py*

Input one video and the script outputs images with plotted skeletons (without player localization, filtering etc), and a json file with the joint coordinates output, which is a list of shape #frames x #detected_persons x #joints x 2 (x and y coordinate)

Usage:

python pose_estimation_script.py [input_file] [output_folder] [-number_frames]

- input_file: path to the input video, must be a video file (.m4v, .mp4, .avi, etc.)
- output_folder: does not need to exist yet
- number_frames: if only a certain number of frames should be processed, specify the number

Example:

```
cd 1_Pose_estimation
python pose_estimation_script.py "../demo_data/example_1.mp4"
"../demo_data/demo_outputs/example_1_output"
```

(This command would process the video *example_1.mp4* in the *demo_data* folder. The outputs are saved in a new folder called *example_1_output* in *demo_data/demo_outputs*. Note that the output is worse than usually because no ROI is used)

2.2 Process videos to joint trajectories

In my framework, a video is read frame by frame and the pose estimation network yields the skeletons of all detected persons. Then the target person is localized and the region of interest for the next frame is calculated. After processing all frames, the output trajectories are smoothed and interpolated.

More in detail, the following steps are executed:

- Pose estimation on each frame –> a list of detected people

- Localize the target player (start position required) –> joint coordinates for each frame for the target person
- swap right and left joints because sometimes in unusual positions they are suddenly swapped
- interpolate missing values
- smooth the trajectories with a lowpass filter
- save the output in a standardized json file for each video, containing a dictionary with 2D coordinates per joint per frame

Usage:

joint_trajectories.py [-h] input_dir output_dir center

Arguments:

- input_dir: folder with video files to be processed
- output_dir: folder where to store the json files with the output coordinates (does not need to exist before)
- center: specify what kind of file is used for specifying the center of the target person: Possible arguments:
  - "../train_data/center_dics.json" for high quality videos (json file with starting position of the target player - either pitcher or batter is filmed in these videos)
  - "datPitcher" for all other kind of videos, if you want to get the joint trajectories for the pitcher
  - "datBatter" for all other kind of videos, if the batter is the target person

Examples:

```
cd 1_Pose_estimation
python joint_trajectories.py ../demo_data/ ../demo_data/demo_outputs
datPitcher

python joint_trajectories.py ../demo_data/ ../demo_data/demo_outputs
datBatter

python joint_trajectories.py
../train_data/high_quality_videos/batter/
../demo_data/demo_outputs ../train_data/center_dics.json

python joint_trajectories.py
../train_data/high_quality_videos/pitcher/
../demo_data/demo_outputs ../train_data/center_dics.json
```

Note: The "center" parameter might be confusing: It refers to the center of the hips of the target person in the first frame, which is required for localizing the target from all detected persons. However, I have tested pose estimation with two different kind of input videos: The database of MLBAM, containing

ten thousands of videos of plays, and 30 high quality videos (downloaded from YouTube). For MLBAM videos, the starting position is given in a *.dat* file for each video (in the same directories). If these videos are used, then dependent on whether the target player is the Pitcher or the Batter, the center argument must be datPitcher or datBatter. Then for each video, the belonging dat file is used to get the start position of the target player. However, for the high quality videos of course no metadata is available. Thus, I just manually labeled the starting position for each video, and put the coordinates in a json file. In this case, the argument must be the path to the json file, which is *../train_data/center_dic*.

2.2.1 Utils for joint trajectories

- The pose estimation is done using the function *handle_one(img)* in the file *pose_estimation_script.py*
- Localization, smoothing and interpolating functions can be found in *data_processing.py*

2.2.2 Convert json files to csv

The *json_to_csv.py* file is used to take all json files of one folder and save them in a csv file instead (better for training models later than loading json files individually every time). In the file, a csv file with metadata and the folder with json files must be specified. Here, all videos were processed already. If you process them again, you can take the output files *cf_pitcher.csv* and *cf_batter.csv* as metadata.

2.3 Player localization (visualization)

A notebook called *player_localization.ipynb* demonstrates the approach for player localization. The IoU approach can be run on a video in which the approach works, in comparison to one in which the target player is lost.

2.4 Filtering (visualization)

A notebook called *visualization_pose_estimation* is used to compare different methods for filling in missing values and smoothing the joint trajectories. The data provided is all processed already. However, for the visualization in this notebook, there is one file with raw data in the *demo_data* folder (*example_1_raw.mp4*)

The (filtered) pose estimation can be plotted on a video. Unfortunately, Quick Time Player on OSX is not able to display the output videos. Use VLC or Elmedia Video Player, or Windows Media Player.

3 Movement classification

This unit of the framework contains files to recognize actions on the field. Basically, it is the last stage of the processing pipeline from videos to motion classes. So first, a video must be processed with the files in the folder *1_Pose_estimation*, such that joint trajectories for **one** player are outputted. For movement classification, these trajectories are used as input to a CNN called MC-CNN that can solve different classification problems.

Here, pose estimation was already run on all available videos and saved in csv files. Thus, for training and testing MC-CNN, the *cf_pitcher.csv* and *cf_batter.csv* files in the *train_data* folder serve as input data and ground truth labels. Since pose estimation is too inaccurate for side-view data, only the joint trajectories of center-field videos are used. 5% are taken as validation data, and the network is trained on the other 95% (again split into training and testing data). For the saved models, the indices saved in *test_indices.npy* were used as test data (indices refer to the rows in the csv files).

3.1 Training

Run the file *classify_movement.py* in order to train a CNN (or other networks) to classify motion. It uses the *run-*, *model-*, and *utils*-files from the main directory to create a Tensorflow graph and train a model. Tasks and data can be specified as arguments.

Usage: classify_movement.py [-h] save_path [-training TRAINING] [-label LABEL]

Arguments:

- save_path: indicates path to save the model if training, or path to the model for restoring if testing
- -training: if training, set True, if testing, set False (default: True)
- -label: "Pitch Type", "Play Outcome" or "Pitching Position (P)" are possible so far

Examples training:

```
cd 2_Movement_classification
python classify_movement.py ../saved_models/pitch_type_new
 -label="Pitch Type"

python classify_movement.py ../saved_models/pitching_position_new
 -label="Pitching Position (P)"
```

3.1.1 Ten-fold-cross validation

To train in 10 fold cross validation, change line 18 in *classify_movement.py* from *"from run_thread import Runner"* to *"from run_10fold import Runner"*. Then, training will be executed ten times, each time on different 90% of the data. The mean accuracies are saved in the file *ten_fold_results.json* in the end.

3.1.2 Parameters

All parameters are set in the *config.py* file.

- Specify if the number of classes for the pitch type (3 superclasses) should be reduced, or if the data should be restricted to 5 players or to one pitching position
- Hyperparameters for the CNN architecture can be set (number of layers, learning rate, number of epochs etc.)
- For completely different ANN architectures, change line 49 in *classify_movement.py*. Available ANNs:
  - "adjustable conv1d": default, MC-CNN - kernel and filter sizes can be changed in in *config.py*
  - "rnn": LSTM with number of stacked cells and hidden units as specified in the *config.py* file
  - "conv1d big": larger CNN with batch normalization (see *model.py* file in main directory)

3.2 Testing

3.2.1 Validation data

Running *classify_movement.py*, tests are automatically run on data from the same csv file (only 95% of this file is taken for training, the other 5% for validation). Make sure that the same configuration (number of players and position included, labels sorted into super classes) as for training is set in the *config.py* file.

Example:

```
cd 2_Movement_classification
python classify_movement.py saved_models/pitch_type
 -label="Pitch Type" -training=False
```

(This tests the pre-trained model on the rows of the csv corresponding to the saved *test_indices*, on the task of classifying the pitch type)

3.2.2 Own data

Use the test file in the main directory to input your own data as a numpy array (no labels required, but can be added for comparison). Use *test.py* with the approriate model from *saved_models* in main folder with your input data.

3.2.3 Available models

The models saved in the folder *saved_models* in the main directory are all trained as explained above: 5% of the csv file were excluded from training and are saved in *train_indices.npy* to serve as validation data. The saved models can thus be tested on this data. The accuracies should be similar to the ones in the table in section 6.6.2 of my thesis.

- pitching_position: 3 classes, trained on all data in *cf_pitcher.csv* for which position labels are available
- play_outcome: Takes data from the *cf_batter.csv* file (batter trajectories)
- pitch_type: All players and positions included (10 classes), with *cf_pitcher.csv*
- pitch_type_5players: Only the data of the five players with most data is used. Since these five pitchers do not use all pitch types, only 7 classes are included
- pitch_type_superclasses: The pitch type classification task is simplified, by sorting the pitch types into superclasses: Fastballs, Breaking Balls and Changeups.
- pitch_type_super_5: Both restrictions from above together: Only the five players with most data and only three classes.

4 Event detection

This folder contains testing- and training files for event detection. Four events can be distinguished: The pitcher's first movement, ball release, the batter's lifting of his leg and the batter's first step towards first base. Test functions for each of these videos can be found in *detect_event.py*. The functions basically return the frame index of an event, given the input data (video frames, joint trajectories etc.). For the experiments on more data though, other notebooks or files are used which will be explained in the following.

4.1 Pitcher's first movement

Experiments are conducted on one game-date (one folder of videos) with different configurations. See the related notebook *pitcher_first_movement.ipynb* for further explanations. Outputs of the experiments are saved in *outputs/first_move_result_dic*.

4.2 Ball release frame

Three different methods to find the ball release frame are presented: Using the joint trajectories to find the moment the arm is highest above the shoulders, training a CNN to learn to recognize the pitcher's stance in images, and running FMO-C to detect the ball.

4.2.1 Arm higher than shoulders

The moment when the arm of the pitcher is highest above his shoulders is taken as the ball release frame. See explanations and visualizations in the related notebook *HS_release.ipynb*.

4.2.2 Stance recognition in images

Training and testing is both implemented in *release_from_image_train.py*. The data is split by specifying some game-dates as test data, and other games as training data. Then, a CNN is trained to distinguish between positive frames (ball release frame) and negative frames (other frames). In the tests, the network is applied on each frame and the one with the highest output is selected as the ball release frame. The results are visualized as box plots of the error distribution.

Usage:

release_from_image_train.py [-h] [-training][-model_save_path]

Optional arguments:

- -training: Set to *"False"* if you want to test the model
- -model_save_path: if training: path to save model, if testing: path to restore model

A pre-trained model is stored in *saved_models/release_model* in the main directory.

Training example:

```
cd 3_Event_detection
python release_from_image_train.py -training=True,
-model_save_path="../saved_models/release_model_new"
```

Testing example: (outputs a boxplot with error)

```
cd 3_Event_detection
python release_from_image_train.py -training=False,
-model_save_path="../saved_models/release_model"
```

4.2.3 Release from ball detection

The ball is detected with FMO-C. Then, with a rough ball speed approximation (2D) and with the distance to the pitcher, it can be concluded when ball release must have occurred. See explanations and tests in the related notebook *release_ball_detection.ipynb*.

4.3 Batter's lifting of the leg

A simple thresholding and maximum approach is employed to find the moment the batter lifts his leg, with his joint trajectories as input. The code is simply a function in the *detect_event.py* file. The experiments are demonstrated in the notebook *batter_movement.ipynb*, where the outputs are visualized together with the batter's first step.

4.4 Batter's first step

For the batter's first step, a LSTM is trained. It learns to find the frame index of the batter's first step, given the joint trajectories of the batter as input.

Use the *batter_first_move_train.py* file to train and test the LSTM. In train_data/batter_runs, the data is directly distinguished between train and test data. Thus, if the argument *training=False* is passed, the corresponding data is processed. Outputs of the test labels are saved in a json file in *outputs/batter_first_move_test_outputs.json*.

Usage:

batter_first_move_train.py [-h] [-training][-model_save_path][-data_path]

- -training: Set to *"False"* if you want to test the model (default: "True" = training)
- -model_save_path: if training: path to save model, if testing: path to restore model
- -data_path: path to data for training and testing

Training example:

```
cd 3_Event_detection
python batter_first_move_train.py -training="True"
-model_save_path="../saved_models/batter_first_step_new"
-data_path="../train_data/batter_runs"
```

Testing example:

A pre-trained model is stored as *saved_models/batter_first_step* in the main directory. It can be tested on the test data with:

```
cd 3_Event_detection
python batter_first_move_train.py -training="False"
-model_save_path="../saved_models/batter_first_step"
-data_path="../train_data/batter_runs"
```

Experiments and visualization of the results of a trained model can be found in the notebook *batter_movement.ipynb* again.

5 Object tracking

5.1 Ball speed

Ball tracking is tested on videos from a public database, filming just the ball trajectory from pitcher to batter (less than one second long).

Experiments are run in a three step process: Firstly the 2D trajectory is estimated with FMO-C, and the outputs are saved in json files in the folder *ball_speed_from_2D/449253* (because the game ID of the processed data is 449253). Secondly, the 2D trajectories are transformed into 3D and the speed is estimated. I did not implement this myself but used code from Prof. Dietrich. Thirdly, the speed outputs are plotted.

5.1.1 2D trajectory

The notebook *ball_speed.ipynb* is used to detect the ball in all videos and save the trajectories as json files. For ball detection, the script *fmo_detection.py* is used. The 2D trajectories are saved in one file for each one-second-video (dictionary with the results for all three cameras). These json files are saved in *ball_speed_from_2D/449253*. See the notebook for further information.

5.1.2 3D coordinates and speed

For this step, C++ code by Prof. Dietrich is used. All code can be found in the folder *ball_speed_from_2D*, which also includes a README with explanations. Unfortunately, the code can only be executed in Windows. To process all videos and output a report containing the speed, simply execute the batch file *449253_from_camera_A* or *449253_from_camera_B* depending on the camera. The output is a "report" csv file that is saved in the folder *ball_speed_from_2D/449253*.

5.1.3 Evaluation

To evaluate the results and plot the error distribution, see the notebook *ball_speed.ipynb* again.

5.2 Bat and glove detection

The bat is detected by a combination of Faster R-CNN and FMO-C. All relevant code and explanations are in the notebook *bat_detection.ipynb*. The notebook uses the Faster R-CNN implementation and models from the Tensorflow Object

Detection API (Huang et al. 2017). The code from the API is stored in the folder *models*.

For this unit, I use the high quality videos in the *train_data* folder in the main directory. There are 44 videos available. I ran Faster R-CNN and FMO-C for each of them and stored the outputs as json file in the folder *outputs*. In each *_fasterrcnn.json* file, the bounding boxes of bat and glove for each frame are saved. In the *_fmoc.json* files, the motion candidates of FMO-C are stored.

In the notebook *bat_detection.ipynb*, FMO-C and Faster R-CNN are merged, tip and base coordinates of the bat are be derived with the wrist position, and the outputs are visualized. They can also be plotted on a video which is saved in the *outputs* as well.

5.2.1 Run FMO-C and Faster R-CNN on all videos

For the experiments though, it is unhandy to run the notebook for each video to get the detection rates. Thus, I created two other files that can process all video at once: Firstly, in *bat_detection.py* FMO-C and Faster R-CNN are applied on all frames of all videos. The outputs are saved in json files as described above.

In order to run FMO-C and Faster R-CNN for all videos, run

```
cd 4_Object_tracking
python bat_detection.py
```

5.2.2 Detection rates

Secondly, *bat_experiments.py* contains the same code as in the notebook for loading the Faster R-CNN and FMO-C results, merging them and displaying the detection rates.

```
cd 4_Object_tracking
python bat_experiments.py
```

prints the results. Since the Faster R-CNN and FMO-C are only evaluated on the frames during the swing, the swing frames for each video had to be found manually. The start and end frame of the swing for each video is saved in *swing_frames_bat.json*. This is loaded in both the notebook and in *bat_experiments.py* in order to calculate the detection rates.

Bibliography

Cao, Zhe, Tomas Simon, Shih-En Wei, and Yaser Sheikh. 2017. "Realtime Multi-Person 2D Pose Estimation Using Part Affinity Fields." In *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, Hi, Usa, July 21-26, 2017*, 1302–10. doi:10.1109/CVPR.2017.143.

Huang, Jonathan, Vivek Rathod, Chen Sun, Menglong Zhu, Anoop Korattikara, Alireza Fathi, Ian Fischer, et al. 2017. "Speed/Accuracy Trade-Offs for Modern Convolutional Object Detectors." In *IEEE Cvpr*.

Rozumnyi, Denys, Jan Kotera, Filip Sroubek, Lukás Novotný, and Jiri Matas. 2017. "The World of Fast Moving Objects." In *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, Hi, Usa, July 21-26, 2017*, 4838–46. doi:10.1109/CVPR.2017.514.