Project AI: Real-time object detection and avoidance for autonomous Nao Robots performing in the Standard Platform League

Rogier van der Weerd, M.Sc. student (13454242), University of Amsterdam

July 13, 2021

Introduction

Since 2010, the Faculty of Science of the University of Amsterdam hosts the Dutch Nao Team (DNT)[1], composed of bachelor and master students, that develops and manages the code base and infrastructure required to participate in the RoboCup Standard Platform League (RSPL)[2]. In this league, autonomous Nao-v6 robots engage in competitive soccer matches. DNT resides in the Intelligent Robotics Lab (IRL)[3], an advanced robotics teaching-and research facility of the University. The RSPL competition provides a unique environment where a range of technologies and skills are integrated into a fully functioning multi-player robotics platform. Such technologies include sensing, motion, localization, computer vision, communication, game-play and behavioral state planning and many more.

DNT's participation in the 2021 RSPL Obstacle Avoidance challenge[4] required the development and implementation of robust real-time object detection and the capability to navigate though a field of obstacles, while walking with a ball. This report describes the approach to this implementation, the results of tests to assess performance as well as potential avenues for further improvement.

A reduced Yolo-v3 model configuration [24] is presented and trained specifically to achieve high performance on robot and ball detection. A basic deployment of this model on a standard Nao-v6 runs at about 700ms, which is far too slow for any high fidelity real-time application. However, by implementing ball and robot model classes that apply Kalman filtering on low-frequency detection signals generated on a separate thread, a stable representation is achieved. We show that this approach is feasible and sufficient for basic obstacle avoidance. F1 scores for detection are significantly improved compared to a legacy detector, from ca. 0.2 to ca. 0.8-0.9. However, we can do better and explore the potential to improve execution time of the detection algorithm. As for the task of navigating through a field of obstacles, a pathfinding module is developed that uses the representation of the field based on the detections. This navigation module can be called by the behavioral engine (a state machine) in any scenario that requires maneuvering through an environment.

1 Starting Point and Objectives

The object avoidance task as formulated in the RSPL challenge[4] requires a robot to walk with a ball through a half-field with up to 5 robots placed as static obstacles and score a goal. The static robots can have different orientations and stances. In order to achieve object avoidance capability, a number of new functionalities are required that need to be integrated into the existing DNT framework. A simplified overview of the modules to be added and their place in the framework is presented in the shaded box in Figure 1.

- First, in order to avoid objects, they will need to be detected with high confidence and projected on a representation of the environment. The current framework contains a Haar feature based cascaded classifier for ball detections [28]. The Haar detector is fast (30 FPS), but inaccurate. It suffers from a high false positive rate and low recall (details are presented in section 3.5), so this needs to be improved.
- Secondly, Ball and Robot models are required, calculated by applying Kalman filters on (potentially noisy) detection signals, in order to make best estimate predictions of object locations with respect to the robot.

[1]Dutch Nao Team `https://www.dutchnaoteam.nl/en/home`
[2]Robocup Standard Platform League.`https://spl.robocup.org/`
[3]Intelligent Robotics Lab, University of Amsterdam `https://robolab.science.uva.nl`
[4]`https://spl.robocup.org/rc2021/#oac`, `https://cdn.robocup.org/spl/wp/2021/01/SPL-Rules-2021.pdf`

- Thirdly, a Navigation module is required to plan a feasible and optimal path to a target position. From that point on, the existing behavior engine can be used to define a behavior that consists of walking with the ball to the first waypoint in the queue.

This work focuses on object detection and navigation capability. Object modelling was developed in a separate project. However, the result of integration tests in real-time game-play scenarios will be presented in section 4.5 of this report.

2 Related Work

Object detection has experienced a tremendous impulse over the last decade [12, 33, 35], driven by research interest and a successful evolution from classical and hand-engineered features to approaches that leverage deep learning. Detection builds on object classification techniques and adds localization of (multiple) objects by predicting bounding boxes. There are many successful approaches that can now achieve near perfect performance on benchmark datasets [35]. Beyond detection lies real-time image segmentation [17] to predict semantic categories on the level of pixels which is also widely and successfully applied in many fields.

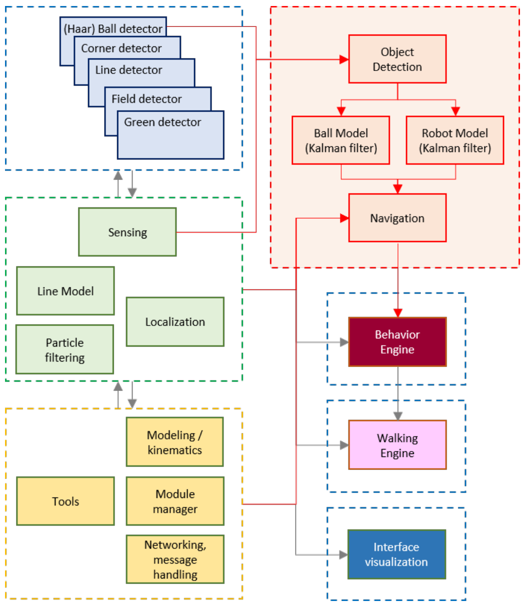


Figure 1: High-level overview of the DNT framework

Application of deep learning in robotics is widespread and many successful applications have been reported in vision, motion, navigation, planning & control using techniques such as deep convolutional neural nets, reinforcement learning and transfer learning [14, 20, 25]. Specifically on object detection and navigation, various teams in the Standard Platform League have reported innovative applications. A comparative survey of Neural Network based approaches to detection from TUHH [13] served as a starting point. Several publications scale down detection networks to achieve acceptable frame rates. [3] introduces xYolo, a scaled down version of Yolo-v3 specifically optimized for ball and goal-post detection. [27] reverts to binary convolutional networks (XNOR) and tests it on a Jetson TX1 device. [21] designed JET-Net, a scaled down model that successfully runs using the optimized JIT compiler developed specifically for the Nao-v6 CPU by the B-Human team from Bremen [29]. [34] reports successful deployment of Yolo-v4 and reduced models on different platforms, including Nao-V6 and a Coral Edge TPU. Progress is also made in dataset generation: [9] offers infrastructure for image tagging and dataset sharing. Benchmark datasets are being developed [8, 19]. An open source generative model for scene generation using Unreal Engine 4 was introduced by [11]. In the area of navigation and path planning, [32] implemented and compares several options to be considered and favors an A*-based algorithm, as does [2].

3 Developing Object Detection capability

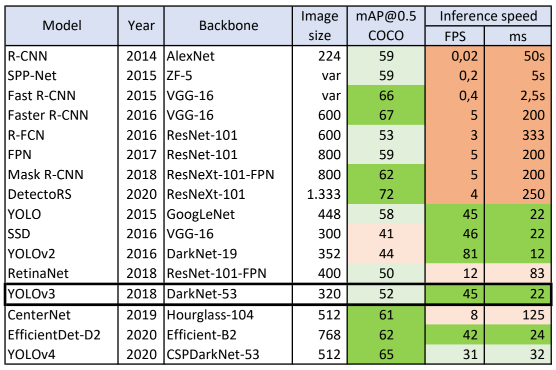
3.1 Starting point and requirements

Given the constraints on computational resources of the robots, most of the current computer vision algorithms deployed on DNT's Nao robots are highly optimized for specific tasks and are based on classical methods such as Canny edge detection and Haar feature-based cascade classifiers. However, these are expected to be insufficient for the challenge and this provides a good opportunity to migrate to Convolutional Neural Network (CNN)-based detection. Various teams in the RSPL have moved in this direction and use cases have been presented at `https://2021.robocup.org/symposium`. There are clear candidates for well suited modern detection algorithms for this task. However, the main challenge will be to minimize model complexity and trade-off model performance with speed and feasibility to run on the Nao platform that has no GPU and heavily used threads on its CPU for normal operation.

The vision modules in the DNT framework have access to 640x480 images in the YUV color space. These images are taken from an upper and lower camera mounted in the head of the robot and are processed at 30 FPS. The objective is to retrieve bounding boxes of objects on the field, such that their location can be derived. There are multiple competing aspects to optimize for. These include *Speed* (low latency for detection), *Accuracy* (high precision and recall), *Robustness* (performance under a wide range of circumstances such as light conditions, orientation of objects, backgrounds, motion blur), *Generalization* (ability to detect a variety of classes beyond balls and robots).

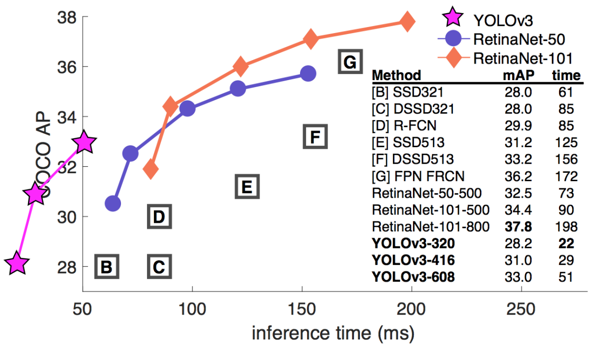
3.2 Survey of methods and method selection

The last 20 years have seen many breakthroughs in object detection techniques, well documented by several surveys [12, 33, 35]. Detection algorithms can be categorized into two-stage (R-CNNs, SPP, FPN, FCN and others) and single-stage (SSD, Yolo, Mask R-CNN, RetinaNet and others) models. Two-stage models split Region of Interest generation from Pooling operations that extract features and specify candidate bounding boxes. This achieves high localization and recognition accuracy, but at the costs of inference speed. Single-stage models combine all operations in one convolutional forward pass which leads to faster inference. In recent years, much effort has been put into developing lightweight networks (examples include SqueezeNet, MobileNet) that can be deployed in resource constrained environments such as mobile and IoT devices. A key element of any detector is the backbone network used to extract features on which to base object classification. These backbones have evolved in complexity and effectiveness and include variants such as AlexNet, VGG, GoogLeNet/Inception, Residual Networks, DarkNet amongst others. Current state-of-the-art approaches[5] are summarized in Figure 2.



Model	Year	Backbone	Image size	mAP@0.5 COCO	Inference speed	
					FPS	ms
R-CNN	2014	AlexNet	224	59	0,02	50s
SPP-Net	2015	ZF-5	var	59	0,2	5s
Fast R-CNN	2015	VGG-16	var	66	0,4	2,5s
Faster R-CNN	2016	VGG-16	600	67	5	200
R-FCN	2016	ResNet-101	600	53	3	333
FPN	2017	ResNet-101	800	59	5	200
Mask R-CNN	2018	ResNeXt-101-FPN	800	62	5	200
DetectoRS	2020	ResNeXt-101	1.333	72	4	250
YOLO	2015	GoogLeNet	448	58	45	22
SSD	2016	VGG-16	300	41	46	22
YOLOv2	2016	DarkNet-19	352	44	81	12
RetinaNet	2018	ResNet-101-FPN	400	50	12	83
YOLOv3	2018	DarkNet-53	320	52	45	22
CenterNet	2019	Hourglass-104	512	61	8	125
EfficientDet-D2	2020	Efficient-B2	768	62	42	24
YOLOv4	2020	CSPDarkNet-53	512	65	31	32

Figure 2: Overview of key detection algorithms[33]



Method	mAP	time
[B] SSD321	28.0	61
[C] DSSD321	28.0	85
[D] R-FCN	29.9	85
[E] SSD513	31.2	125
[F] DSSD513	33.2	156
[G] FPN FRCN	36.2	172
RetinaNet-50-500	32.5	73
RetinaNet-101-500	34.4	90
RetinaNet-101-800	**37.8**	198
YOLOv3-320	28.2	**22**
YOLOv3-416	31.0	29
YOLOv3-608	33.0	51

Figure 3: Performance envelope[24]

For real-time detection tasks, an optimum is sought between accuracy[6] and inference speed, visualized in a performance envelope, see Figure 3. The Yolo algorithm, since its first formulation in 2016 [22] has consistently pushed the boundary in this envelope. There are currently five generations of the Yolo algorithm [4, 22, 23, 24], the details of which are well summarized by [30]. Key aspects and improvements between Yolo generations will be presented in the next section.
Yolo is a clear candidate of choice as a generic and high performing detection algorithm. It is preferred over highly optimized algorithms for specific classes given the ability to include additional classes in the future that can be relevant for the DNT framework such as goal posts, penalty markers, different robot stances, etc. As a basis for our application, variants of the the third generation Yolo-v3 are considered. The key reason for this is the maturity of this version, with well established implementation frameworks and portability options. Yolo-v3 contains all critical improvements needed for our purposes.

3.3 Key principles used by Yolo

Bounding box predictions One of the key innovations that the first version Yolo-v1 [22] introduced was the *Convolutional implementation of sliding windows* for bounding box prediction using (1x1) kernels. The use of (1x1) kernels to substitute fully connected layers had already been demonstrated in 2013 (dubbed network-in-network [16]). [26] introduced Overfeat: instead of running forward propagation on all individual sliding windows of the input image independently, these windows can be combined into one forward propagation computation and share the computation in the common regions of the image. An illustration with more details is provided in Figure 26 (Appendix F.1). Yolo-v1 built on this approach to predict exact bounding boxes. It splits an image into grid cells ($g_x \times g_y$), assigns object boxes to specific grid cells based on the center pixel of the

[5]Care should be taken when interpreting performance metrics as these depend highly on the choice of evaluation dataset and hardware. Only relative performance will be considered within well executed benchmark studies

[6]Usually expressed in mean average precision for detection tasks, this will be discussed in the next section

box and defines an output vector $[p_c, b_x, b_y, w_x, w_y, c_1, ..., c_k]^T$, with k number of classes. p_c can be interpreted as the 'objectness': the probability that there is an object in the bounding box[7]. (b_x, b_y) defines the center of the bounding box, and (w_x, w_y) its width and height, relative to the grid cell. c_i represents the conditional probability of class i, given that there is an object. Figure 4 shows the end result: a convolutional network with appropriate architecture can map an image of shape $(w \times h \times 3)$ to a volume with output vectors for each grid cell $(g_x \times g_y \times (5 + k))$.

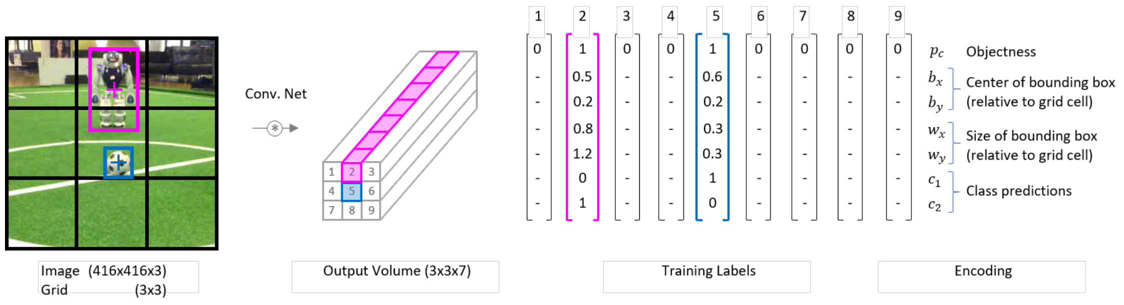
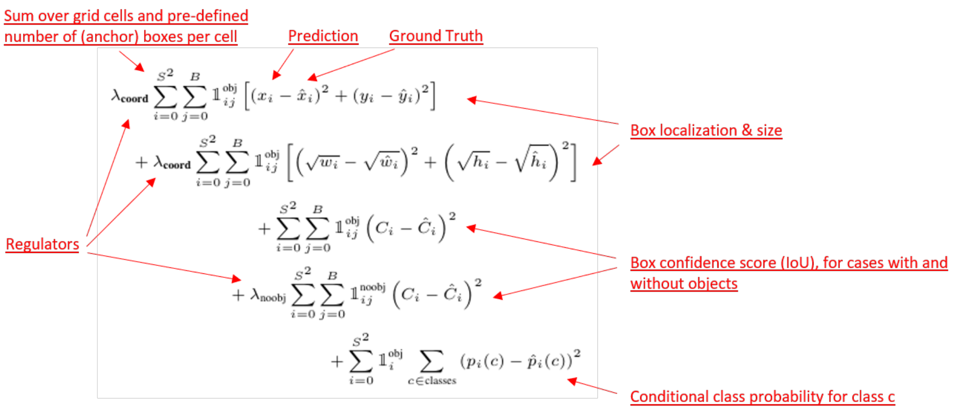

Conv. Net
1
2
3
4
5
6
7
8
9
p_c Objectness
b_x b_y Center of bounding box (relative to grid cell)
w_x w_y Size of bounding box (relative to grid cell)
c_1 c_2 Class predictions
Image (416x416x3) Grid (3x3)
Output Volume (3x3x7)
Training Labels
Encoding

Figure 4: Illustration of transformation from image to Yolo output vector encodings (Image adapted from [18])

Loss function With these definitions, Yolo defines a loss function used for training the network, see Figure 5. The loss consists of four terms representing i) how close is the predicted bounding box to the ground truth, ii) how close are width and height of the box compared to the ground truth, iii) how close is the objectness score to the ground truth label (either 0 or 1), and iv) how close is the predicted conditional class probability to the ground truth class.


Sum over grid cells and pre-defined number of (anchor) boxes per cell
Prediction
Ground Truth
Box localization & size
Regulators
Box confidence score (IoU), for cases with and without objects
Conditional class probability for class c

$$\lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left[(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2 \right]
+ \lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left[\left(\sqrt{w_i} - \sqrt{\hat{w}_i}\right)^2 + \left(\sqrt{h_i} - \sqrt{\hat{h}_i}\right)^2 \right]
+ \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left(C_i - \hat{C}_i\right)^2
+ \lambda_{\text{noobj}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{noobj}} \left(C_i - \hat{C}_i\right)^2
+ \sum_{i=0}^{S^2} \mathbb{1}_{i}^{\text{obj}} \sum_{c \in \text{classes}} (p_i(c) - \hat{p}_i(c))^2$$

Figure 5: Yolo loss function (annotations made on original image from [22])

Non-max suppression When an output volume such as the example in Figure 4 is obtained, it can predict multiple detections of the same object with mid-points in different grid cells. *Non-max suppression* is used to discard all bounding box predictions below a p_c (objectness) threshold (this is a hyperparameter, usually set at 0.6). Next, all remaining predictions are processed in decreasing order based on p_c. For overlapping bounding boxes (based on an Intersection-over-Union, or IoU above some threshold, usually set at 0.5), the prediction with the highest objectness score is kept and all others are discarded (suppressed). Non-max suppression is performed independently on all predicted classes to avoid any unintended suppression due to interference of bounding boxes between classes.

Anchor boxes Anchor boxes were introduced in Yolo-v2 [23] and enable detection of multiple objects per grid cell. Key idea is to pre-define archetype bounding box sizes, let's say three per grid cell. The output tensor shown in Figure 4, as well as the ground truth labels, will now contain three vectors of size (7×1) per grid cell, resulting in a volume of $(g_x \times g_y \times b(5 + k))$, with k the number of classes and b the number of anchor boxes. K-means clustering can be used to define anchor boxes (also considered as priors for the class predictions) that are most suitable to the different classes in the training set. Some technicalities are involved to map network output to bounding box coordinates relative to the anchor boxes, details can be found in [23].

[7]Or, alternatively: a prediction of the Intersection-over-Union (IoU) of the bounding box with the ground truth

Multi-scale learning Among several incremental improvements in Yolo-v3 [24], scale pyramids were introduced in the Yolo architecture that enable detections at different resolutions by the same network. This is particularly relevant for our detection task (think of soccer balls that need to be detected at different distances). In addition, Yolo-v3 changed the backbone of the network to Darknet-53, containing 53 convolutional layers with residual connections.

Evaluation For the detection task, the most appropriate evaluation metric is Mean Average Precision (mAP), defined using Intersection over Union for all predicted bounding boxes and classes in a given image. mAP is calculated based on a choice of (a range of) confidence thresholds, details are provided in Appendix F.2.

Other improvements After Joseph Redmon, who worked on the first three generations of Yolo, withdrew from further development, other researchers have continued the work and have introduced Yolo-v4 and v5. Improvements were made to the network architecture introducing dense blocks, spatial pyramid pooling blocks, a path aggregation network and a range of additional 'tricks' in Yolo-v4 [4]. Another group ported the Yolo framework to Pytorch[8] and claimed the Yolo-v5 label around the same time, but this has caused some controversy given the limited extent of the improvement and the lack of thorough review and publication of results.

The overall pipeline involved when training and deploying Yolo is shown in Figure 6. Note that images need to be pre-processed (scaling to fit the input dimensions and optional augmentation during training) and the network's output requires post-processing (non-max suppression) both for training and inference.

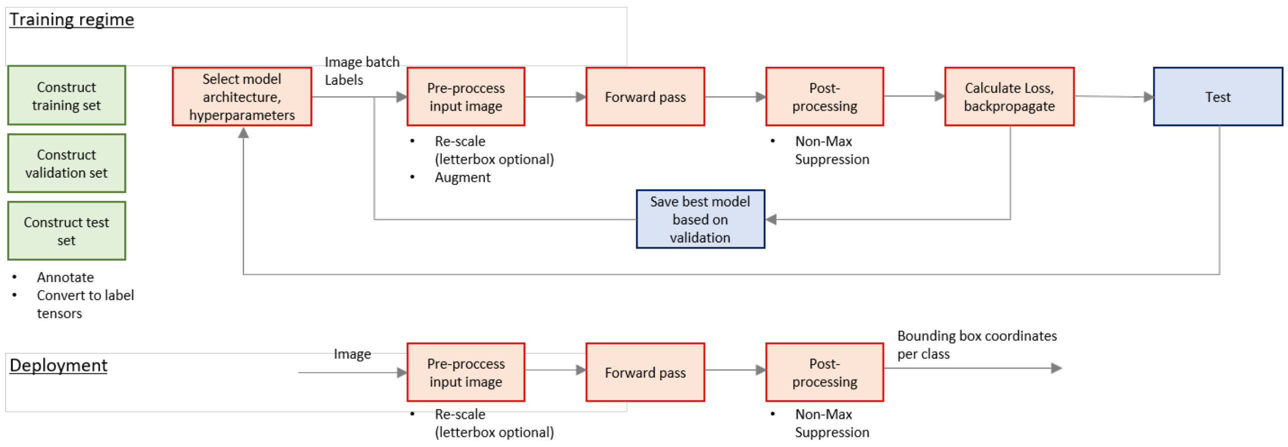


Figure 6: Summary of the pipeline used for training and deploying Yolo

3.4 Training and testing Tiny Yolo-v3/3L

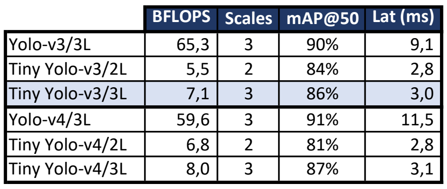
Dataset generation A critical component of obtaining a good detector is availability of a dataset that is large and diverse enough to enable the network to learn, generalize and ultimately predict detections in new unseen scenes. The ball and robot classes are very specific to the RSPL and therefore, pre-trained models are not sufficient for this task[9]. In 2018, a 'Bit-Bots' team from the University of Hamburg released an open source online platform[10] for collaborative image labeling specifically for the RoboCup Soccer [9]. A number of annotated datasets were extracted from this database and added to the training dataset. In addition, new data was generated by recording live scenes of DNT robots, in order to enrich the train dataset with more diverse robot poses (in particular crouched robots). A diverse[11] set of scenes were recorded to serve as validation data. Although one could bootstrap an automated annotation tool for the specific classes, images were annotated by hand using the Bit-Bots platform. All extracted annotations were verified. The train and validation datasets consist of ca. 7,500 and 600 images, respectively. Appendix C presents statistics and example images from the datasets.

Model selection The Darknet framework[12] was used to train and evaluate Yolo v3 and v4 models. For each version, three configurations were considered: the full scale model with prediction on three scales ('Yolo-vX/3L'), the original reduced model that uses two scales ('Tiny Yolo-vX/2L') and a the reduced model with

[8]`https://github.com/ultralytics/yolov5`
[9]Fine-tuning of pre-trained weights can be used when features extraction is expected to be effective on new classes
[10]`https://imagetagger.bit-bots.de`
[11]In terms of light conditions, number/location/stance of robots on the field, distance of objects, richness of background imagery
[12]`https://github.com/AlexeyAB`, forked from the original framework by Joseph Redmon, maintained by Alexey Bochkovsky

three scales ('Tiny Yolo-vX/3L'). Key hyperparameters that were kept constant include the image input size (416x416)[13], anchor box dimensions[14], data augmentation, batch size, and optimizer settings. Figure 7 presents the results. It is clear that the reduced models[15] are well suited for this task with Tiny Yolo-v3/3L offering a good middle ground: limited in complexity (i.e. high inference speed) and mAP performance of 86%, close to the maximum achieved. This model was chosen for use on the robots and its performance will be presented next. The detailed model architecture of Tiny Yolo-v3/3L is provided in Appendix A.

Training The Tiny Yolo-v3/3L model was trained using the Darknet framework, compiled to run on an Aurora-R9 work station with an Intel Core i7 9700 (8 cores/16 threads) and Nvidia 2080 Ti GPU (11 GB) with CUDA 10 and driver 415. Training for 20k steps, which corresponds to ca. 170 epochs (7k images in the training set, batch size 64) takes ca. 110 minutes. All parameters used are presented in Appendix A.3. A batch size of 64 is used, learning rate of 0.001 (scheduled to scale down by 0.1 at steps 10k and 15k), an SGD optimizer is used with momentum 0.9 and weight decay 5e-4. Data augmentation is used with parameters or saturation (1.5), exposure (1.5) and hue (0.1)[17]. The progression of the loss and the performance on the validation set (mean average precision) during training is shown in Figure 8. In this particular run, the best model was found at around step 9,500 (epoch 80).



	BFLOPS	Scales	mAP@50	Lat (ms)
Yolo-v3/3L	65,3	3	90%	9,1
Tiny Yolo-v3/2L	5,5	2	84%	2,8
Tiny Yolo-v3/3L	7,1	3	86%	3,0
Yolo-v4/3L	59,6	3	91%	11,5
Tiny Yolo-v4/2L	6,8	2	81%	2,8
Tiny Yolo-v4/3L	8,0	3	87%	3,1

Figure 7: Performance of Yolo-v3 and v4 variants on validation dataset[16]

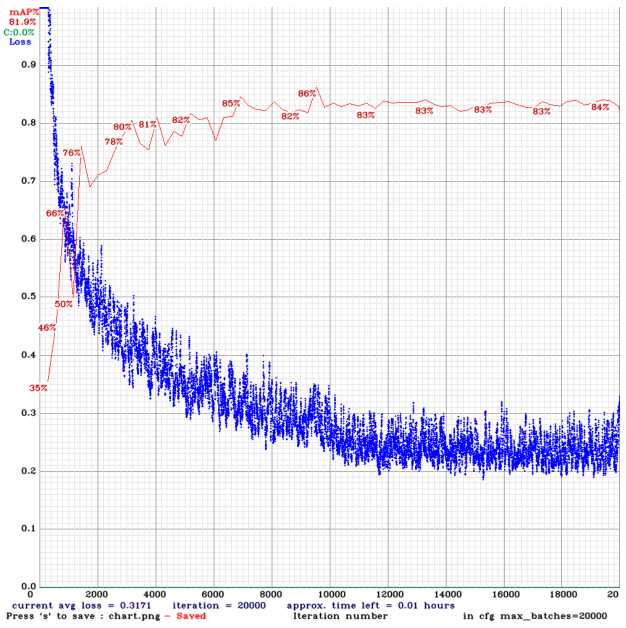
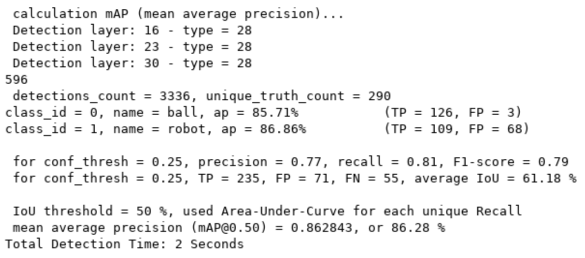


Figure 8: Chart of the progression of training loss and performance on validation data during training



```
 calculation mAP (mean average precision)...
 Detection layer: 16 - type = 28
 Detection layer: 23 - type = 28
 Detection layer: 30 - type = 28
596
 detections_count = 3336, unique_truth_count = 290
class_id = 0, name = ball, ap = 85.71%           (TP = 126, FP = 3)
class_id = 1, name = robot, ap = 86.86%          (TP = 109, FP = 68)

 for conf_thresh = 0.25, precision = 0.77, recall = 0.81, F1-score = 0.79
 for conf_thresh = 0.25, TP = 235, FP = 71, FN = 55, average IoU = 61.18 %

 IoU threshold = 50 %, used Area-Under-Curve for each unique Recall
 mean average precision (mAP@0.50) = 0.862843, or 86.28 %
Total Detection Time: 2 Seconds
```

Figure 9: Screen output of the results of Darknet model evaluation on the validation dataset

Off-line Evaluation Darknet offers a facility to evaluate a trained model on test data. Results are shown in Figure 9. The overall Mean Average Precision on the test data for a confidence threshold of 0.5 is $mAP_{@0.50}$ = 86%, with nearly equal performance on ball and robot classes. The overall F1 score is 0.79. These are promising results given the diversity and, at times, complex detection scenes (with high levels of motion blur, partially occluded or distant objects). A verification and visual inspection of results will be presented based on deployment on the robot in the next section.

3.5 Deployment of the Yolo detector on the Robot

Embedding detection functionality in DNT's Robotics software The DNT code base, written in C++17, constitutes a framework in which all functionality required for autonomous operation of the robots is integrated. For the task of object detection and object avoidance, the key module groups that need to be

[13]Lower resolution harmed detection of smaller/more distant objects
[14]These were obtained by running k-means clustering on the train dataset, using 3 anchor boxes per grid cell
[15]As a comparison: the full Yolo-v3 model has 62M parameters, Tiny Yolo-v3/3L has 9M
[16]Latency recorded using a Nvidia 2080 Ti GPU
[17]For definitions and visualizations, see `https://www.ccoderun.ca/darkmark/DataAugmentationColour.html`

developed are *perception* (transforming real-time imagery to relevant signals to be used by other modules), *object modelling* (maintaining an accurate and current representation of the environment: robot and ball positions on the field), *navigation* (planning feasible and optimal paths towards a target position) and the *behavior engine* (overall logic to inform the robot's decisions, in particular as input for motion control).

Implementation details There are various options available for implementing the Yolo model. The initial approach taken leverages the functionality offered by OpenCV's Deep Neural Net (dnn) module[18], which can load and execute Darknet encoded models through its API. The DNT code that uses this API is available in a private repository, excerpts of key code snippets are included in Appendix B.1. Execution time of inference using the Tiny Yolo-v3/3L model on the Nao Robot's CPU[19] is about 700ms, equivalent to ca. 1.4 FPS. This is too slow for real-time detection, with the robots operating at a frame rate of ca. 30 FPS. Options for speedup will be discussed later. In order to have a functioning baseline detector, two steps were taken:

1. New `ballModel` and `robotModel` classes were created, that apply Kalman filtering on the low-frequency Yolo detection signals. The signals are infrequent but very reliable and hence this probabilistic approach enables a stable and usable estimate of ball and robot positions. Implementation details are out of scope of this report.

2. Detection frequency is particularly important tracking faster ball movement. The DNT framework already used a Haar feature based cascaded classifier for ball detections [28, 31]. Although this detector runs at 30 FPS, it is not very accurate, with high false positive[20] and false negative rates. A `Mixed Ball Signal Generation` algorithm was implemented to combine the speed of the Haar classifier with the precision of Yolo, shown in Algorithm 1. The key idea is to validate Haar detections whenever infrequent, periodic Yolo detections are executed. If the Haar classifier is in validated state and detects positively in subsequent frames (without Yolo executing), we assume it is still valid, as long as the time between detections is below a threshold. Otherwise, we consider the Haar detection a False Positive and discard it.

Algorithm 1 Mixed Ball Signal Generation

```
1: while Robot is active do                                              ▷ Embedded in main operating loop
2:   function GetBallSignal(timestamp)                                    ▷ All access to data members handled in class
3:     cam ← pointer to latest camera sensor output
4:     haar ← pointer to latest Haar ball classifier output
5:     yolo ← pointer to Yolo class object
6:     timeSinceLastYoloRun ← currentTime − yolo.lastYoloRun
7:
8:     if haar.timeSinceLastDetection > 1s or timeSinceLastYoloRun > 1s then
9:       ballBoundingBoxes, robotBoundingBoxes ← yolo.RunDetection(cam)
10:      yolo.lastYoloRun ← update
11:      haar.validated ← false
12:      for all b ∈ ballBoundingBoxes do
13:        if haar.ballRectange overlaps with b then                    ▷ Discards False Positives of Haar classifier
14:          haar.validated ← true
15:        end if
16:      end for
17:      return ballBoundingBoxes, robotBoundingBoxes
18:    else
19:      if haar.validated then                                         ▷ Fast Haar detections are used if validated in an earlier cycle
20:        return haar.ballRectangle
21:      end if
22:    end if
23:  end function
24: end while
```

Evaluation on the robots The approach described above was tested by recording a test match on DNT's home field and analyzing a randomly selected sequence of ca. 600 consecutive frames of game-play. Performance on Ball and Robot detection can now be evaluated separately. For ball detection, we are mixing results of two detectors, each of which can yield a True Positive (TP), False Positive (FP), True Negative (TN) or (False Negative) result. Thus there are $4 \times 4 = 16$ possible outcomes. Examples of the most common outcomes are shown in Figure 10. Detailed analysis is provided in appendix D. Figure 11 presents the key resulting evaluation metrics. A marked performance improvement is observable when the Yolo detector is used. Yolo's ball detection precision is nearly perfect and recall improves from 19% (Haar classifier) to 70% (mixed detection). Recall on robot detection is 76%. Overall F1 scores improve from 0.3 (Haar classifier) to 0.8 and above when using the Yolo detector.

[18]`https://docs.opencv.org/4.3.0/d6/d0f/group__dnn.html`, (The DNT framework currently uses OpenCV v.4.3.0)
[19]Intel Atom E3845 Quad Core @ 1.91 GHz, 4GB DDR3
[20]Especially on robots: the Haar classifier often falsely detects a ball in the regions of knee and torso of the robots

Visual inspection of the result indeed confirms many false positive detection errors by the Haar classifier in regions of the robot, see Figure 20 (Appendix D.1). Many ball detections are missed by Haar, especially at higher distances. Yolo, on the other hand, is robust against motion blur and successfully detects at different distances (scales) as is expected given the 3-layer scale pyramid used in Tiny Yolo-v3/3L.

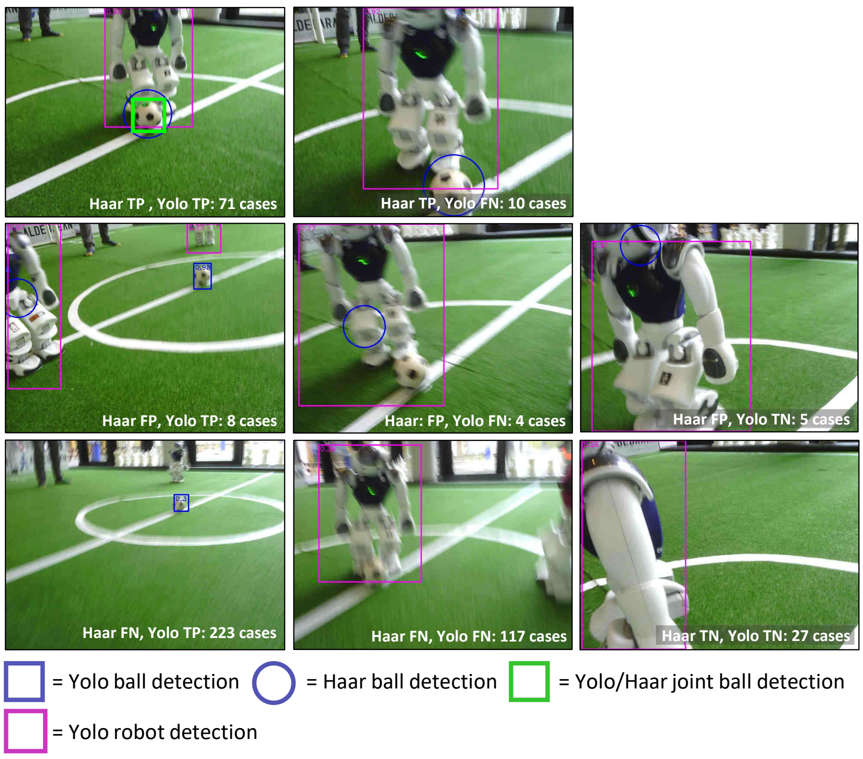
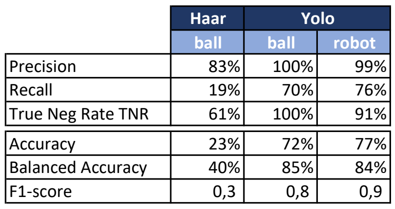

Haar TP , Yolo TP: 71 cases
Haar TP, Yolo FN: 10 cases
Haar FP, Yolo TP: 8 cases
Haar: FP, Yolo FN: 4 cases
Haar FP, Yolo TN: 5 cases
Haar FN, Yolo TP: 223 cases
Haar FN, Yolo FN: 117 cases
Haar TN, Yolo TN: 27 cases
= Yolo ball detection
= Haar ball detection
= Yolo/Haar joint ball detection
= Yolo robot detection

Figure 10: Illustration of potential outcomes of mixed Haar classifier and Tiny Yolo-v3/3L based ball detections



	Haar	Yolo	
	ball	ball	robot
Precision	83%	100%	99%
Recall	19%	70%	76%
True Neg Rate TNR	61%	100%	91%
Accuracy	23%	72%	77%
Balanced Accuracy	40%	85%	84%
F1-score	0,3	0,8	0,9

Figure 11: Detection performance of Haar vs. Tiny Yolo-v3/3L

3.6 Options for acceleration

The challenge that comes from deploying neural networks on resource limited devices has seen much research interest in recent years [6, 10, 15]. There are various options to consider for speeding up inference times and reducing energy use. They are briefly mentioned here, but implementation and experimentation was not performed within the scope of this project.

Code and compiler optimization Naturally, all redundant operations should be eliminated from the code that calls the forward pass of the network. This includes unnecessary scaling, color space transformations and class instantiations. A first step to consider would be to use the JIT compiler[21] optimized for the Nao-v6 CPU [29], shown to achieve up to an order of magnitude faster inference times on complex networks.

Hardware acceleration A second route to consider in the context of RSPL would be the use of specialized hardware such as TPUs or camera devices with integrated specialized processing units. As an experiment, the trained Tiny Yolo-v3/3L model was run on a Luxonis Oak-1 camera using the DepthAI framework[22], achieving 25 FPS. However, these hardware additions are not allowed under the current RSPL rules.

Model complexity reduction, pruning and quantization We have seen the benefit of model complexity reduction with the Tiny Yolo variants. Once defined, a baseline network architecture may still contain a large amount of redundancy [10]. Pruning addresses the issue of redundant weights in the network. Complexity is reduced by eliminating weights that do not (significantly) affect prediction outcomes for a given task. Quantization addresses the unnecessary precision at which calculations are performed. Significant speedups can be achieved by moving to lower precision (INT8, in some cases even up to binary) and readjusting (clipping and recalculating) weights to maintain model performance. A lot of investment has been going into the development of seamless pruning and quantization pipelines[23], some specifically targeting Yolo models[24].

[21]`https://github.com/bhuman/CompiledNN`
[22]`https://docs.luxonis.com/projects/hardware/en/latest/pages/BK1096.html`, `https://docs.luxonis.com/en/latest/`
[23]All major deep learning frameworks offer optimization modules
[24]Notably `https://neuralmagic.com/blog/benchmark-yolov3-on-cpus-with-deepsparse`

4 Developing Navigation and Pathfinding Capability

4.1 Starting point and requirements

The DNT framework had no navigation or waypoint system in place before participating in the Object Avoidance challenge. The basic robot behavior during game-play consisted of <find ball>→<walk to ball>→<align with goal>→<walk to goal>. This basic behavior can be expanded by building three new functionalities:

1. Maintain an active representation of objects on the field (object modelling)
2. Based on own position and target position, find the best feasible path and transform this path to a discrete set of waypoints
3. Rewrite the <align with goal> and <walk to goal> behaviors into more flexible <align with waypoint> and <walk to waypoint> behaviors (in which, of course, some point in the goalbox can also be defined as a waypoint).

Within the scope of this project, active waypoint generation was developed (functionality 2), while other members of the team developed object modelling and walk to waypoint behavior. The basic requirement of waypoint generation is to plan paths to a predefined position through a field with robots as obstacles in various stances. These paths should be formed such that they i) are feasible to walk while kicking a ball, implying that the fewer and shallower the turns required, the better; ii) retain safe distance margins from robots; iii) minimize the time required to walks the path.

4.2 Survey of methods

Mobile robot path planning is a well studied problem and a myriad of effective approaches are possible [1, 5, 7], roughly categorized into global planning (further divided into approaches using graph search, random sampling or intelligent bionic algorithms) and local planning (ranging from basic graph-based approaches to the use of potential fields, fuzzy logic, particle algorithms, etc.). In the context of the SPL, the German team 'HULKs' team from Hamburg University of Technology has reported on their study of an A* based approach in 2019 [32]. However, this was evaluated offline and was not ported to a robot for testing.
The complexity of the task is limited and it is expected that a coarse grid will be sufficient given field dimensions (9m × 6m), robot dimensions (safe distance radius is assumed $40cm$) and limited number of robots on the field. The approach taken is an alteration of Dijkstra's shortest path algorithm, which applies a weighting of edges in a graph with nodes representing field cells. The approach does not rely on a choice of heuristic required by A*.

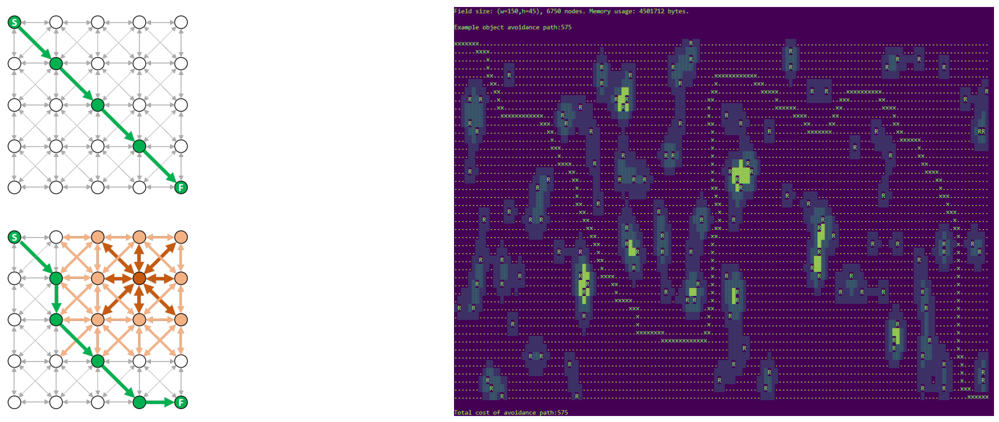
4.3 Shortest path implementation using a cost landscape

The implementation of Dijkstra's shortest path algorithm for robot navigation on the soccer field starts with a custom graph class[25]. The field is divided into a graph of nodes and edges $G(N, E)$, with each node connected bidirectionally in 8 directions. An empty field will have edges with cost 1 (directions N,S,W,E) or $\sqrt{2}$ (directions NE,SE,SW,NW). The class implements a parameterized placement of obstacles: all edges originating from the obstacle node are assigned a peak cost c_{max}, which attenuates over adjacent nodes $\alpha^i \cdot c_{max}$, $i \in 1, ..n$ based on a the reach of the cost mountain n. This assignment is additive, meaning that if robots are near each other, their cost mountains will interfere and create a saddle. Hence, a 'cost landscape' is created. The implementation of the shortest path algorithm is provided in Appendix B.2. It uses the priority queue class from C++'s Standard Library and runs in $O(NlogN + E)$.

4.4 Testing

The approach is visualized in Figure 12 for a cost mountain with $c_{max} = 100$, reach of $n = 1$ and decay of $\alpha = 0.75$. Figure 12b shows a test case on the console with a large grid and high density of obstacles. The landscape approach ensures safe distances by navigating over the saddles. Note that this represents an overly complex toy example case. Parameters were tuned for realistic scenarios following the SPL rulebook[26]. The parameters and test scenarios for the prototype module are provided in Appendix E.

[25]The interface of this class is provided in Appendix B.2. The full prototype code can be found at `https://github.com/rvdweerd/containers_GraphClassImplementation/tree/DNTgraph` and its implementation in the DNT framework at `https://github.com/IntelligentRoboticsLab/DNT2017/tree/worldwide2021-obj-avoid/source/dnt/tools/navigation` (private)
[26]`https://cdn.robocup.org/spl/wp/2021/01/SPL-Rules-2021.pdf`


S
F
Field size: (w=150,h=45), 6750 nodes. Memory usage: 4501712 bytes.
Example object avoidance path:575
Total cost of avoidance path:575

(a) Visualization of a basic cost landscape in a graph

(b) Illustrative example of navigating through an additive cost landscape created by randomly placing robots

Figure 12: Demonstration of edge-weighted shortest path finding using a cost landscape on a graph

4.5 Deployment on the robot

The code was ported to the robot by adding a navigation module to the DNT framework. For testing and debugging purposes, a visual representation of the process was added to the DNT interface, see Figure 13. The scene represents the start of a game as seen from the upper camera of the test robot. Detection bounding boxes are marked, including the transformed coordinates from image space (the bottom centers of the bounding boxes) to world space (relative to the robot in its own coordinate system). This represents the estimate of the position of the objects on the field. These are derived using inverse kinematics of the robot. The right panel presents a top view the field[27] with the test robot position and orientation marked in black, and detected field robots, clipped to the nearest grid cells in the small red circles. The path to the goal (in this example straightforward, although it navigates in between the field robots) is indicated by the blue line and marked waypoint grid cells.

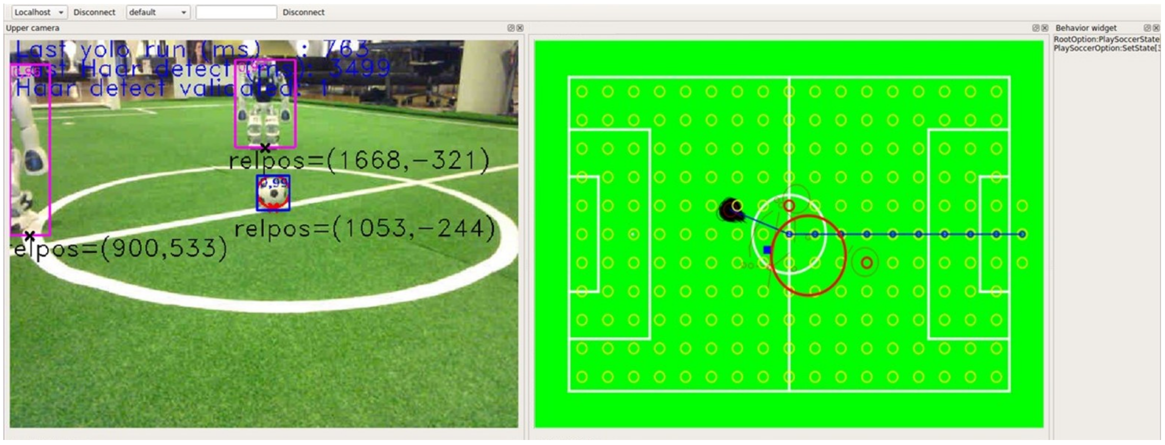


Figure 13: Scene from integration test, shown as a replay on the DNT Interface (legend in Appendix E)

Conclusion

The 2021 RoboCup Object Avoidance challenge offered a good opportunity to extend object detection and navigation/pathfinding capabilities in the codebase of the Dutch Nao Team. Within the scope of this project, object avoidance capability was successfully developed, as demonstrated by integration tests.

The choice for Tiny Yolo-v3/3L as the detection model was motivated by an optimization of speed and accuracy, in combination with maturity and hence availability of frameworks and easy portability. The deployment on Nao-v6 using OpenCV's dnn module is slow, with inference times around 700ms. However, it was shown that by using probabilistic estimation (Kalman filtering), stable and feasible representations of ball and robot positions can be achieved. These form the input of a weighted shortest path algorithm that yields waypoints, leading to the capability to maneuver to a target position on the field, while avoiding detected objects.

[27] A legend of the symbols in this image is provided in Appendix E

The quality and quantity of the training dataset was crucial for obtaining a robust detection result. A significant effort was made to create a diverse and effective dataset. Further improvement can be explored in two areas: generalization of detection and speedup of the detection algorithm. The Tiny Yolo-v3/3L model can be trained to detect additional landmarks on the field such as goalposts and penalty markers. This may be used for improved localization and navigation. Significant speedup is expected by applying sparsification (pruning and quantization) of the models, as well as as integrating a compiler optimization library that specifically targets the Nao-v6 hardware.

References

[1] Zeyad Abd Algfoor, Mohd Shahrizal Sunar, and Hoshang Kolivand. A comprehensive study on pathfinding techniques for robotics and video games. *International Journal of Computer Games Technology*, 2015, 2015.

[2] Rahib H Abiyev, Murat Arslan, Irfan Gunsel, and Ahmet Cagman. Robot pathfinding using vision based obstacle detection. In *2017 3rd IEEE International Conference on Cybernetics (CYBCONF)*, pages 1–6. IEEE, 2017.

[3] Daniel Barry, Munir Shah, Merel Keijsers, Humayun Khan, and Banon Hopman. Xyolo: A model for real-time object detection in humanoid soccer on low-end hardware. In *2019 International Conference on Image and Vision Computing New Zealand (IVCNZ)*, pages 1–6. IEEE, 2019.

[4] Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. Yolov4: Optimal speed and accuracy of object detection. *arXiv preprint arXiv:2004.10934*, 2020.

[5] Kuanqi Cai, Chaoqun Wang, Jiyu Cheng, Clarence W De Silva, and Max Q-H Meng. Mobile robot path planning in dynamic environments: a survey. *arXiv preprint arXiv:2006.14195*, 2020.

[6] Yu Cheng, Duo Wang, Pan Zhou, and Tao Zhang. A survey of model compression and acceleration for deep neural networks. *arXiv preprint arXiv:1710.09282*, 2017.

[7] Márcia M Costa and Manuel F Silva. A survey on path planning algorithms for mobile robots. In *2019 IEEE International Conference on Autonomous Robot Systems and Competitions (ICARSC)*, pages 1–7. IEEE, 2019.

[8] Roberto Fernandes, Walber M Rodrigues, and Edna Barros. Dataset and benchmarking of real-time embedded object detection for robocup ssl. *arXiv preprint arXiv:2106.14597*, 2021.

[9] Niklas Fiedler, Marc Bestmann, and Norman Hendrich. Imagetagger: An open source online platform for collaborative image labeling. In *Robot World Cup*, pages 162–169. Springer, 2018.

[10] Amir Gholami, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W Mahoney, and Kurt Keutzer. A survey of quantization methods for efficient neural network inference. *arXiv preprint arXiv:2103.13630*, 2021.

[11] Timm Hess, Martin Mundt, Tobias Weis, and Visvanathan Ramesh. Large-scale stochastic scene generation and semantic annotation for deep convolutional neural network training in the robocup spl. In *Robot World Cup*, pages 33–44. Springer, 2017.

[12] Licheng Jiao, Fan Zhang, Fang Liu, Shuyuan Yang, Lingling Li, Zhixi Feng, and Rong Qu. A survey of deep learning-based object detection. *IEEE access*, 7:128837–128868, 2019.

[13] Chris Kahlefendt. A comparison and evaluation of neural network-based classification approaches for the purpose of a robot detection on the nao robotic system. Master's thesis, Technische Universität Hamburg-Harburg, 2017.

[14] Artúr István Károly, Péter Galambos, József Kuti, and Imre J Rudas. Deep learning in robotics: Survey on model structures and training strategies. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 51(1):266–279, 2020.

[15] Tailin Liang, John Glossner, Lei Wang, and Shaobo Shi. Pruning and quantization for deep neural network acceleration: A survey. *arXiv preprint arXiv:2101.09671*, 2021.

[16] Min Lin, Qiang Chen, and Shuicheng Yan. Network in network. In Yoshua Bengio and Yann LeCun, editors, *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*, 2014.

[17] Shervin Minaee, Yuri Y Boykov, Fatih Porikli, Antonio J Plaza, Nasser Kehtarnavaz, and Demetri Terzopoulos. Image segmentation using deep learning: A survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2021.

[18] Andrew Ng. Coursera course on convolutional neural networks. `https://www.coursera.org/learn/convolutional-neural-networks`. Accessed: May-2021.

[19] Simon O'Keeffe and Rudi Villing. A benchmark data set and evaluation of deep learning architectures for ball detection in the robocup spl. In *Robot World Cup*, pages 398–409. Springer, 2017.

[20] Harry A Pierson and Michael S Gashler. Deep learning in robotics: a review of recent research. *Advanced Robotics*, 31(16):821–835, 2017.

[21] Bernd Poppinga and Tim Laue. Jet-net: real-time object detection for mobile robots. In *Robot World Cup*, pages 227–240. Springer, 2019.

[22] Joseph Redmon, Santosh Kumar Divvala, Ross B. Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, pages 779–788. IEEE Computer Society, 2016.

[23] Joseph Redmon and Ali Farhadi. YOLO9000: better, faster, stronger. In *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, HI, USA, July 21-26, 2017*, pages 6517–6525. IEEE Computer Society, 2017.

[24] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. *arXiv preprint arXiv:1804.02767*, 2018.

[25] Javier Ruiz-del Solar, Patricio Loncomilla, and Naiomi Soto. A survey on deep learning methods for robot vision. *arXiv preprint arXiv:1803.10862*, 2018.

[26] Pierre Sermanet, David Eigen, Xiang Zhang, Michaël Mathieu, Rob Fergus, and Yann LeCun. Overfeat: Integrated recognition, localization and detection using convolutional networks. In Yoshua Bengio and Yann LeCun, editors, *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*, 2014.

[27] Susanto Susanto, Febri Alwan Putra, and Riska Analia. Xnor-yolo: The high precision of the ball and goal detecting on the barelang-fc robot soccer. In *2020 3rd International Conference on Applied Engineering (ICAE)*, pages 1–5. IEEE, 2020.

[28] Duncan ten Velthuis. Nao detection with a cascade of boosted weak classifier based on haar-like features. Bachelor's thesis, Universiteit van Amsterdam, 2014.

[29] Felix Thielke and Arne Hasselbring. A jit compiler for neural network inference. In *Robot World Cup*, pages 448–456. Springer, 2019.

[30] Do Thuan. Evolution of yolo algorithm and yolov5: the state-of-the-art object detection algorithm. Bachelor's thesis, University of Oulu, 2021.

[31] Paul Viola and Michael Jones. Rapid object detection using a boosted cascade of simple features. In *Proceedings of the 2001 IEEE computer society conference on computer vision and pattern recognition. CVPR 2001*, volume 1, pages I–I. Ieee, 2001.

[32] Felix Warmuth. A graph based path planning approach for the robocup standard platform league. Research project, Hamburg University of Technology, 2019.

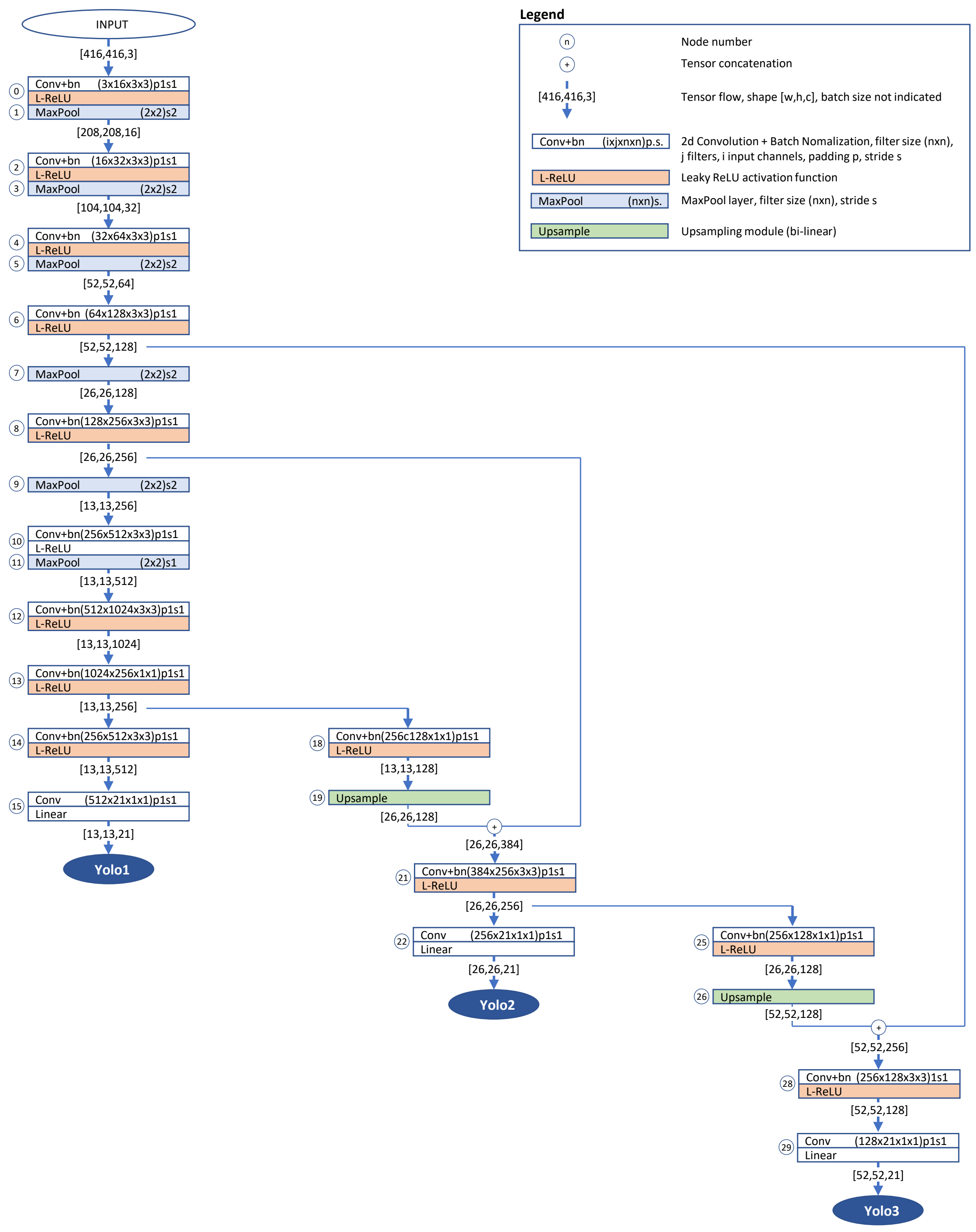
[33] Syed Sahil Abbas Zaidi, Mohammad Samar Ansari, Asra Aslam, Nadia Kanwal, Mamoona Asghar, and Brian Lee. A survey of modern deep learning based object detection models. *arXiv preprint arXiv:2104.11892*, 2021.

[34] Simon O'Keeffe ZhengBai Yao, Will Douglas and Rudi Villing. Faster yolo-lite: Faster object detection on robot and edge devices. In *2021 RoboCup International Symposium*. RSPL, 2021.

[35] Zhengxia Zou, Zhenwei Shi, Yuhong Guo, and Jieping Ye. Object detection in 20 years: A survey. preprint arXiv:1905.05055, 2019.

A Tiny Yolo-v3, 3 layers model specification

A.1 Tiny Yolo-v3, 3 layers model architecture



Legend
n Node number
+ Tensor concatenation
[416,416,3] Tensor flow, shape [w,h,c], batch size not indicated
Conv+bn (ixjxnxn)p.s. 2d Convolution + Batch Nomalization, filter size (nxn), j filters, i input channels, padding p, stride s
L-ReLU Leaky ReLU activation function
MaxPool (nxn)s. MaxPool layer, filter size (nxn), stride s
Upsample Upsampling module (bi-linear)
INPUT
[416,416,3]
0 Conv+bn (3x16x3x3)p1s1
L-ReLU
1 MaxPool (2x2)s2
[208,208,16]
2 Conv+bn (16x32x3x3)p1s1
L-ReLU
3 MaxPool (2x2)s2
[104,104,32]
4 Conv+bn (32x64x3x3)p1s1
L-ReLU
5 MaxPool (2x2)s2
[52,52,64]
6 Conv+bn (64x128x3x3)p1s1
L-ReLU
[52,52,128]
7 MaxPool (2x2)s2
[26,26,128]
8 Conv+bn(128x256x3x3)p1s1
L-ReLU
[26,26,256]
9 MaxPool (2x2)s2
[13,13,256]
10 Conv+bn(256x512x3x3)p1s1
L-ReLU
11 MaxPool (2x2)s1
[13,13,512]
12 Conv+bn(512x1024x3x3)p1s1
L-ReLU
[13,13,1024]
13 Conv+bn(1024x256x1x1)p1s1
L-ReLU
[13,13,256]
14 Conv+bn(256x512x3x3)p1s1
L-ReLU
[13,13,512]
15 Conv (512x21x1x1)p1s1
Linear
[13,13,21]
Yolo1
18 Conv+bn(256c128x1x1)p1s1
L-ReLU
[13,13,128]
19 Upsample
[26,26,128]
+
[26,26,384]
21 Conv+bn(384x256x3x3)p1s1
L-ReLU
[26,26,256]
22 Conv (256x21x1x1)p1s1
Linear
[26,26,21]
Yolo2
25 Conv+bn(256x128x1x1)p1s1
L-ReLU
[26,26,128]
26 Upsample
[52,52,128]
+
[52,52,256]
28 Conv+bn (256x128x3x3)1s1
L-ReLU
[52,52,128]
29 Conv (128x21x1x1)p1s1
Linear
[52,52,21]
Yolo3

Figure 14: Model architecture for the baseline Tiny Yolo-v3 model with three output scales

A.2 Tiny Yolo-v3, 3 layers trainable weights

Node	Module	Weight tensor	#Params
0	Conv2d.weight	[16, 3, 3, 3]	432
	BatchNorm2d.weight	[16]	16
	BatchNorm2d.bias	[16]	16
2	Conv2d.weight	[32, 16, 3, 3]	4,608
	BatchNorm2d.weight	[32]	32
	BatchNorm2d.bias	[32]	32
4	Conv2d.weight	[64, 32, 3, 3]	18,432
	BatchNorm2d.weight	[64]	64
	BatchNorm2d.bias	[64]	64
6	Conv2d.weight	[128, 64, 3, 3]	73,728
	BatchNorm2d.weight	[128]	128
	BatchNorm2d.bias	[128]	128
8	Conv2d.weight	[256, 128, 3, 3]	294,912
	BatchNorm2d.weight	[256]	256
	BatchNorm2d.bias	[256]	256
10	Conv2d.weight	[512, 256, 3, 3]	1,179,648
	BatchNorm2d.weight	[512]	512
	BatchNorm2d.bias	[512]	512
12	Conv2d.weight	[1024, 512, 3, 3]	4,718,592
	BatchNorm2d.weight	[1024]	1,024
	BatchNorm2d.bias	[1024]	1,024
13	Conv2d.weight	[256, 1024, 1, 1]	262,144
	BatchNorm2d.weight	[256]	256
	BatchNorm2d.bias	[256]	256
14	Conv2d.weight	[512, 256, 3, 3]	1,179,648
	BatchNorm2d.weight	[512]	512
	BatchNorm2d.bias	[512]	512
15	Conv2d.weight	[21, 512, 1, 1]	10,752
	Conv2d.bias	[21]	21
18	Conv2d.weight	[128, 256, 1, 1]	32,768
	BatchNorm2d.weight	[128]	128
	BatchNorm2d.bias	[128]	128
21	Conv2d.weight	[256, 384, 3, 3]	884,736
	BatchNorm2d.weight	[256]	256
	BatchNorm2d.bias	[256]	256
22	Conv2d.weight	[21, 256, 1, 1]	5,376
	Conv2d.bias	[21]	21
25	Conv2d.weight	[128, 256, 1, 1]	32,768
	BatchNorm2d.weight	[128]	128
	BatchNorm2d.bias	[128]	128
28	Conv2d.weight	[128, 256, 3, 3]	294,912
	BatchNorm2d.weight	[128]	128
	BatchNorm2d.bias	[128]	128
29	Conv2d.weight	[21, 128, 1, 1]	2,688
	Conv2d.bias	[21]	21
		Total number of weights	9,003,087

Table 1: Number of trainable weights of the baseline Tiny Yolo-v3 model

A.3 Tiny Yolo-v3, 3 layers configuration file using the Darknet standard

```
[net]
# Training
batch=64
subdivisions=4
width=416
height=416

channels=3
momentum=0.9
decay=0.0005
angle=0
saturation = 1.5
exposure = 1.5
hue=.1

learning_rate=0.001
burn_in=250
max_batches = 20000
policy=steps
steps=10000,15000
scales=.1,.1

# CUSTOM
max_chart_loss=1
mosaic=0

[convolutional]
batch_normalize=1
filters=16
size=3
stride=1
pad=1
activation=leaky

[maxpool]
size=2
stride=2

[convolutional]
batch_normalize=1
filters=32
size=3
stride=1
pad=1
activation=leaky

[maxpool]
size=2
stride=2

[convolutional]
batch_normalize=1
filters=64
size=3
stride=1
pad=1
activation=leaky

[maxpool]
size=2
stride=2

[convolutional]
batch_normalize=1
filters=128
size=3
stride=1
pad=1
activation=leaky

[maxpool]
size=2
stride=2

[convolutional]
batch_normalize=1
filters=256
size=3
stride=1
pad=1
activation=leaky

[maxpool]
size=2
stride=2

[convolutional]
batch_normalize=1
filters=512
size=3
stride=1
pad=1
activation=leaky

[maxpool]
size=2
stride=1

[convolutional]
batch_normalize=1
filters=1024
size=3
stride=1
pad=1
activation=leaky

###########

[convolutional]
batch_normalize=1
filters=256
size=1
stride=1
pad=1
activation=leaky

[convolutional]
batch_normalize=1
filters=512
size=3
stride=1
pad=1
activation=leaky

[convolutional]
size=1
stride=1
pad=1
filters=21
activation=linear

[yolo]
mask = 6,7,8
anchors =  4,  7,   7, 15,  13, 25,
          25, 42,  41, 67,  75, 94,
          91,162, 158,205, 250,332
classes=2
num=9
jitter=.3
ignore_thresh = .7
truth_thresh = 1
random=1

[route]
layers = -4

[convolutional]
batch_normalize=1
filters=128
size=1
stride=1
pad=1
activation=leaky

[upsample]
stride=2

[route]
layers = -1, 8

[convolutional]
batch_normalize=1
filters=256
size=3
stride=1
pad=1
activation=leaky

[convolutional]
size=1
stride=1
pad=1
filters=21
activation=linear

[yolo]
mask = 3,4,5
anchors =  4,  7,   7, 15,  13, 25,
          25, 42,  41, 67,  75, 94,
          91,162, 158,205, 250,332
classes=2
num=9
jitter=.3
ignore_thresh = .7
truth_thresh = 1
random=1

[route]
layers = -3

[convolutional]
batch_normalize=1
filters=128
size=1
stride=1
pad=1
activation=leaky

[upsample]
stride=2

[route]
layers = -1, 6

[convolutional]
batch_normalize=1
filters=128
size=3
stride=1
pad=1
activation=leaky

[convolutional]
size=1
stride=1
pad=1
filters=21
activation=linear

[yolo]
mask = 0,1,2
anchors =  4,  7,   7, 15,  13, 25,
          25, 42,  41, 67,  75, 94,
          91,162, 158,205, 250,332
classes=2
num=9
jitter=.3
ignore_thresh = .7
truth_thresh = 1
random=1
```

B C++ Implementation details

Note: All code is available on DNT's private repository on Github, access upon request.[28]

B.1 Darknet model implementation using OpenCV's dnn module

```cpp
#include <opencv2/core.hpp>
#include <opencv2/dnn.hpp>
#include <opencv2/objdetect.hpp>

class mYoloDetector : public Module {
public:
  /**
   * Constructor.
   **/
  mYoloDetector() ;

  /**
   * Default destructor.
   **/
  ~mYoloDetector() = default;

  /**
   * Detection function, runs the pre-loaded yolo model on the current frame's top view camera
     image.
   * @param ball_bounding_boxes placeholder for ball detections, will be filled when there are
     ball detections
   * @param robot_bounding_boxes placeholder for field robot detections, will be filled when
    there are field robot detections
   **/
  void runDetection(std::vector<boundingBox>& ball_bounding_boxes, std::vector<boundingBox>&
    robot_bounding_boxes);

private:
  /**
   * Instantiation of the yolo detection model. Loads Darknet tiny-yolo_v3 model configuration
     file
   * and pre-trained weights in opencv's dnn format
   */
  cv::dnn::Net model_;
  cv::dnn::DetectionModel detection_model_ = cv::dnn::DetectionModel(model_);
};
```

Listing 1: Excerpt of header file `m_yolo_detector.h` defining a dnn-based Yolo detector

```cpp
#include <opencv2/imgcodecs.hpp>

#include "m_yolo_detector.h"
#include "tools/log.h"
#include "tools/system_tools.h"

mYoloDetector::mYoloDetector() {
  // Load configuration file
  Configuration::Data configuration = Configuration::load("yolo_detector.toml");

  // Read in parameters
  yolo_confidence_threshold_ = configuration["yolo_confidence_threshold_"];
  yolo_iou_threshold_ = configuration["yolo_iou_threshold_"];
  yolo_max_time_between_detections_ = configuration["yolo_max_time_between_detections_"];
  yolo_max_time_between_haar_detections_ = configuration["
    yolo_max_time_between_haar_detections_"];

  // Float 32 tiny yolo v3
  yolo_model_configuration_path_ = SystemTools::getRootDir() + "/config/yolo_detector/yolov3-
    tiny_31.cfg";
  yolo_model_weights_path_ = SystemTools::getRootDir() + "/config/yolo_detector/best86_yuv.
    weights";

  model_ = cv::dnn::readNetFromDarknet(yolo_model_configuration_path_,
    yolo_model_weights_path_);
```

[28] https://github.com/IntelligentRoboticsLab/DNT2017/tree/master/source/dnt/modules/perception/yolo_detector, https://github.com/IntelligentRoboticsLab/DNT2017/tree/worldwide2021-obj-avoid/source/dnt/tools/navigation, https://github.com/IntelligentRoboticsLab/DNT2017/blob/worldwide2021-obj-avoid/source/dnt/modules/modelling/m_navigation.h, https://www.dutchnaoteam.nl

```

  detection_model_ = cv::dnn::DetectionModel(model_);
  detection_model_.setInputParams(1 / 255.0, cv::Size2i(416, 416), cv::Scalar(0, 0, 0), true,
    false);
}
```
Listing 2: Key code snippet (not complete) of the class constructor used in `m_yolo_detector.cpp` to instantiate the trained Yolo detector

```
void mYoloDetector::runDetection(std::vector<boundingBox>& ball_bounding_boxes,
                                 std::vector<boundingBox>& robot_bounding_boxes) {
  // Following line is only used for timing yolo execution when testing
  //std::chrono::high_resolution_clock::time_point t1 = std::chrono::high_resolution_clock::
    now();

  cv::Mat frame = camera_sensor_representation_->image_.getYUVImage();
  std::vector<int> classids;
  std::vector<float> confidences;
  std::vector<cv::Rect> boxes;
  detection_model_.detect(frame, classids, confidences, boxes, yolo_confidence_threshold_,
                          yolo_iou_threshold_);

  // Process yolo detections, if any
  if (classids.size() > 0) {
    // Fill bounding box vectors for detected balls (classid=0, Darknet model) and robots
    // (classid=1, Darknet model)
    for (size_t i = 0; i < classids.size(); ++i) {
      if (classids[i] == boundingBox::ClassName::Ball) {
        ball_bounding_boxes.push_back(boundingBox(confidences[i], boundingBox::ClassName::Ball
    , boxes[i], true));
        calculateRelativePositions(ball_bounding_boxes.back());

      } else if (classids[i] == boundingBox::ClassName::Robot) {
        robot_bounding_boxes.push_back(boundingBox(confidences[i], boundingBox::ClassName::
    Robot, boxes[i], true));
        calculateRelativePositions(robot_bounding_boxes.back());
      }
    }
  }
}
```
Listing 3: Key code snippet (not complete) used in `m_yolo_detector.cpp` to execute a detection

B.2 Graph class used for path finding

```
#include <string>
#include <set>
#include <map>
#include <queue>

class SimpGraph
{
private:
  typedef long long int LL;
  struct Arc;
  struct Node
  {
    Node(LL n)
      :
      id(n),
      coord({ n>>32, (n<<32)>>32 })
    {
      name = "(" + std::to_string(coord.first) + "," + std::to_string(coord.second) + ")";
    }
    std::string name;
    LL id;
    std::pair<LL, LL> coord;
    std::set<Arc*> arcs;
    float height = 0;
    int distanceToTop = 0;
  };
  struct Arc
  {
    Arc(Node* s, Node* f, float c)
      :
      start(s),
      finish(f),
      cost(c)
    {
    }
    Node* start;
    Node* finish;
    float cost;
  };
public:
  SimpGraph(int width, int height) : fieldWidth_(width), fieldHeight_(height) {
    InitializeAsGrid({ width,height });
  }
  ~SimpGraph() {
    for (auto n : nodes) {
      delete n;
      n = nullptr;
    }
    for (auto a : arcs) {
      delete a;
      a = nullptr;
    }

  };
  void PrintAdjacencyList();
  void PlotPath(std::vector<std::pair<LL, LL>> path);
  void ResetGridAndPlaceObstacleHills(std::vector<std::pair<int, int>> obstacleGridLocations,
    float peakCost, int spread);
  float findShortestPath(std::pair<LL, LL> startcell, std::pair<LL, LL> endcell, std::vector<
    std::pair<LL, LL>>& pathCoords, std::vector<std::string>& pathNames);

private:
  void InitializeAsGrid(std::pair<int, int> WxH);
  void AddNode(std::pair<LL, LL> pos);
  void AddNode(LL id);
  void AddArc(Node* start, Node* finish, int cost);
  void AddOneWayConnection(std::pair<LL, LL> n1, std::pair<LL, LL> n2, int c);
  void AddTwoWayConnection(std::pair<LL, LL> n1, std::pair<LL, LL> n2, int c);
  void PlaceObstacleCellOnly(std::pair<LL, LL> node, float cost);
  void PlaceObstacleHillUsingBFS(std::pair<LL, LL> startcell, float initial_cost, int
    hill_size);
  void PlaceObstacleHillUsingBFS(Node* startnode, float initial_cost, int hill_size);
  void priceEdgesUsingBFS(int hill_size, float cost_step);
  void VisitedPrintFunction(Node* node);
  void ResetGrid();
  void DFS(std::pair<LL, LL> startcell);
  void DFS(Node* startnode);
  void BFS(std::pair<LL, LL> startcell);
  void BFS(Node* startnode);
  void visitUsingDFS(Node* node);
  void visitUsingBFS();
  struct GreaterPathLength;
  std::vector<Arc*> findShortestPath(Node* start, Node* finish);
  static int getPathCost(const std::vector<Arc*>& path);

private:
  int fieldWidth_;
  int fieldHeight_;
  std::vector<std::pair<int,int>> obstacleCenters;
  std::map<LL, Node*> nodeMap;
  std::set<Node*> nodes;
  std::set<Arc*> arcs;
  std::set<Node*> visited;
  std::queue<Node*> tovisit;
};
```

Listing 4: Interface of the Graph class used to perform path finding using Dijkstra's shortest path algorithm with cost landscapes

B.3 Implementation of Dijkstra’s shortest path algorithm

```cpp
std::vector<SimpGraph::Arc*> SimpGraph::findShortestPath(Node* start, Node* finish)
{
  std::vector<Arc*> arcPath;
  std::priority_queue< std::vector<Arc*>, std::vector<std::vector<Arc*>>, GreaterPathLength>
    queue;
  std::map<LL, float> fixed;
  while (start != finish)
  {
    if (fixed.find(start->id) == fixed.end())
    {
      fixed[start->id] = getPathCost(arcPath);
      for (Arc* arc : start->arcs)
      {
        if (fixed.find(arc->finish->id) == fixed.end())
        {
          arcPath.push_back(arc);
          queue.push(arcPath);
          arcPath.pop_back();
        }
      }
    }
    if (queue.size() == 0)
    {
      arcPath.clear();
      return arcPath;
    }
    arcPath = queue.top(); queue.pop();
    start = arcPath[arcPath.size() - 1]->finish;
  }
  return arcPath;
}

float SimpGraph::getPathCost(const std::vector<Arc*>& path)
{
  float cost = 0;
  for (Arc* arc : path)
  {
    cost += arc->cost;
  }
  return cost;
}

struct GreaterPathLength;
{
  bool operator()(const std::vector<Arc*>& lhs, const std::vector<Arc*>& rhs) const
  {
    return getPathCost(lhs) > getPathCost(rhs);
  }
};
```

Listing 5: Dijksta’s Shortest Path algorithm adjusted for navigating through a cost landscape

C Datasets

C.1 Datasets - Statistics

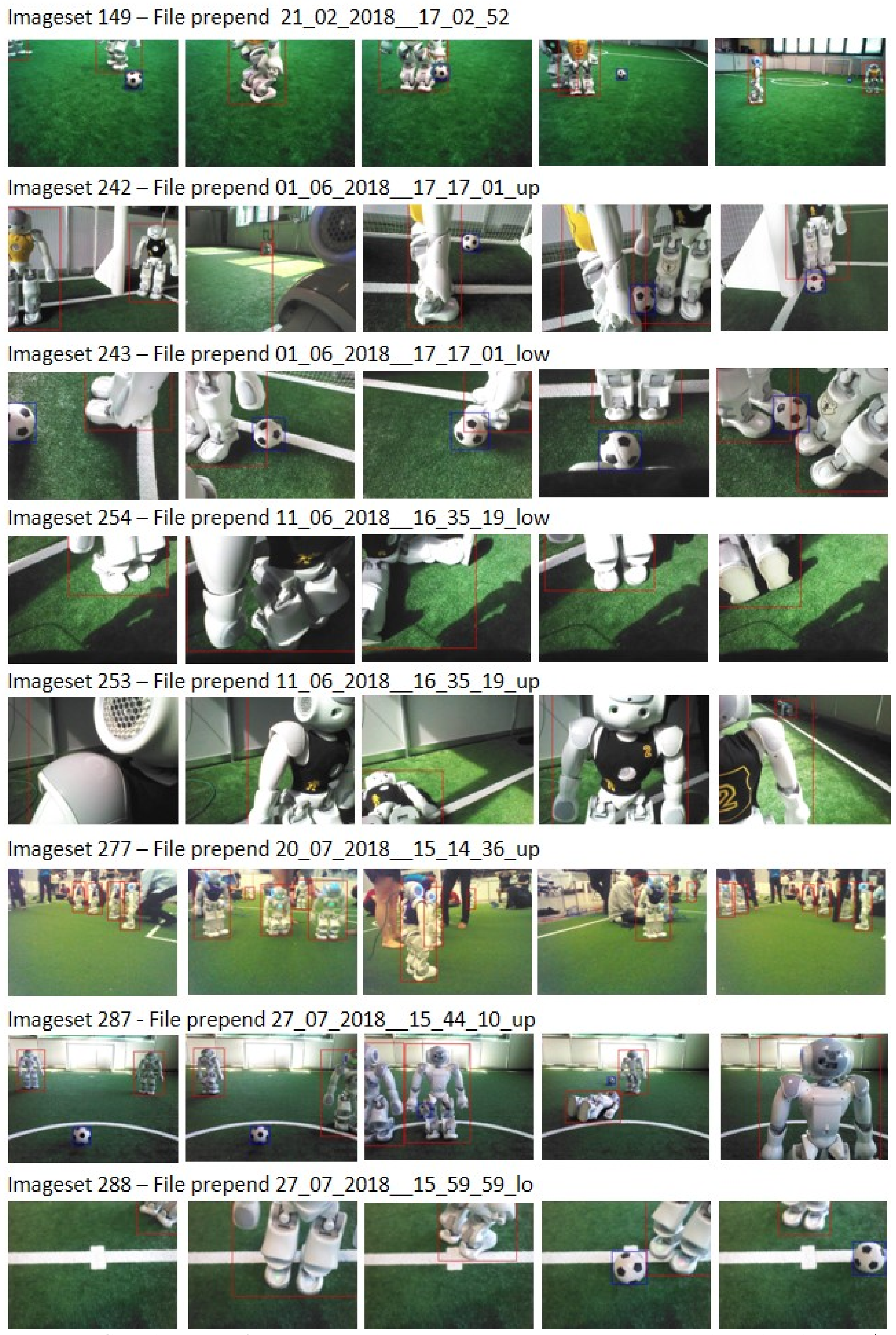
				# Annotations		
Dataset ID	File Prepend	Description	#Images	Ball	Robot	Source
149	21_02_2018__17_02_52	Some scenes of a game with three robots. NaoDevils	329	329	552	bit-bots.de
242	01_06_2018__17_17_01	CloseRobots_1_6_18_Loki_1	558	249	309	bit-bots.de
243	01_06_2018__17_17_01	Close Robots lower NaoDevils	249	249	280	bit-bots.de
253	11_06_2018__16_35_19	Close robots bright light (upp cam) NaoDevils	445	399	556	bit-bots.de
254	11_06_2018__16_35_19	Close robots bright light (low cam) NaoDevils	446	435	429	bit-bots.de
277	20_07_2018__15_14_36	Mixed robots, RoboCup 2018, Montreal	1,104	548	2,603	bit-bots.de
287	27_07_2018__15_44_10	Evaluation set of two robots. Upp cam. NaoDevils	842	380	1,576	bit-bots.de
288	27_07_2018__15_59_59	Evaluation set of two robots. Low cam. NaoDevils	842	844	842	bit-bots.de
289	27_07_2018__16_11_04	EvalLog_StandingRobotMovingHead_upper	637	640	922	bit-bots.de
290	27_07_2018__16_16_30	EvalLog_StandingRobotMovingHead_lower	638	630	641	bit-bots.de
309	22_08_2018__14_58_20	Robots get up motion. Upper cam. NaoDev arena	562	0	562	bit-bots.de
310	22_08_2018__15_04_13	Robots get up motion. Lower cam. NaoDev arena	562	562	562	bit-bots.de
DNT06	frames_saves1	Working day save data, crouched robots	94	112	108	DNT
DNT07	frames_saves2	Working day save data, crouched robots	38	55	62	DNT
DNT08	frames_saves3	Working day save data, crouched robots	188	243	254	DNT
		Total	7,534	5,675	10,258	

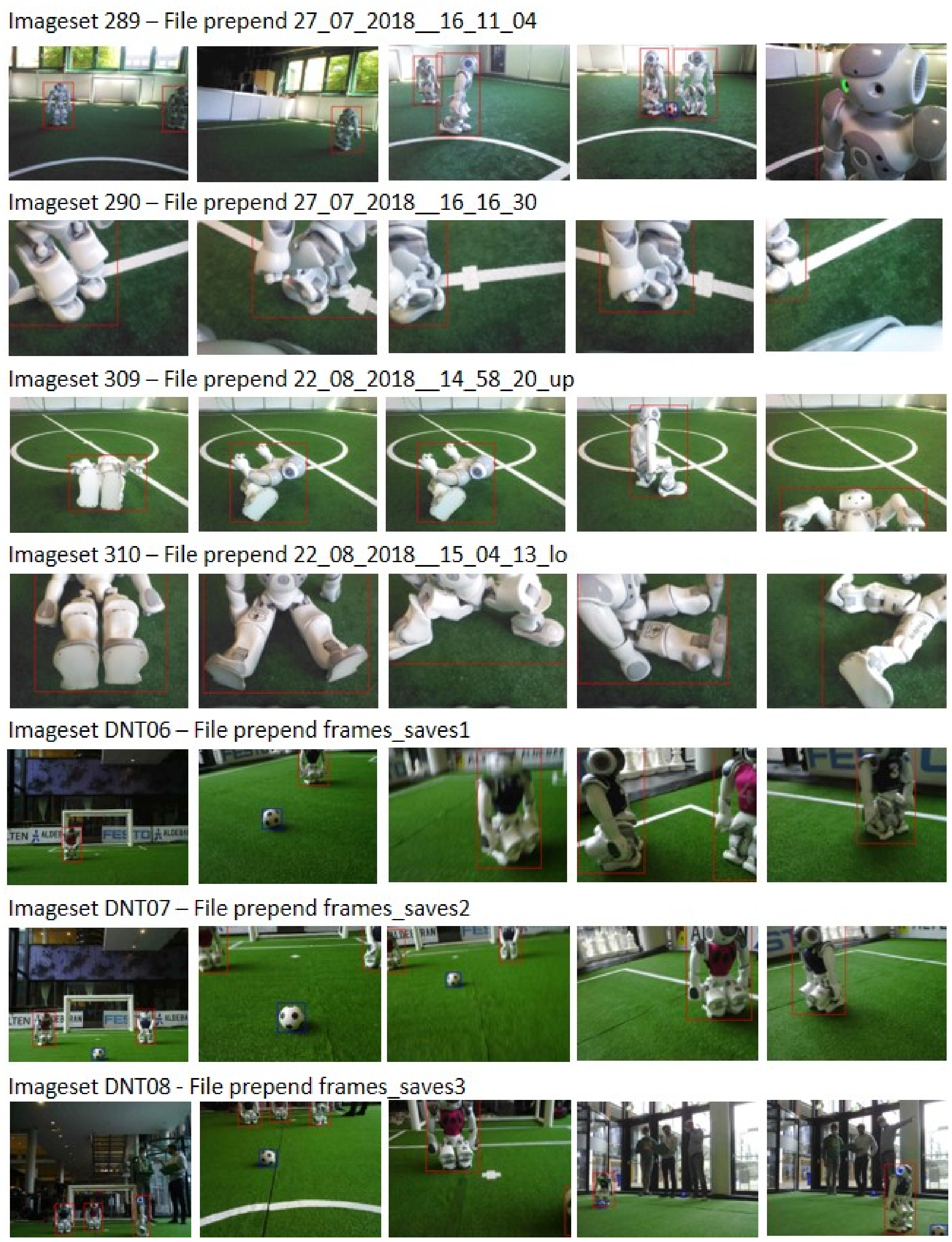
Table 2: Train datasets, key statistics

				# Annotations		
Dataset ID	File Prepend	Description	#Images	Ball	Robot	Source
DNT01	frameset_a	Normal light conditions, robots playing with one ball	208	143	349	DNT
DNT02	frameset_b	Dark light conditions, distant robots	13	8	16	DNT
DNT03	frameset_c	High contrast conditions, focus on balls	148	101	53	DNT
DNT04	frameset_d	Normal light conditions, focus on balls with background noise	190	138	318	DNT
DNT05	frameset_e	Multiple robots playing with one ball	37	27	62	DNT
		Total	596	417	798	

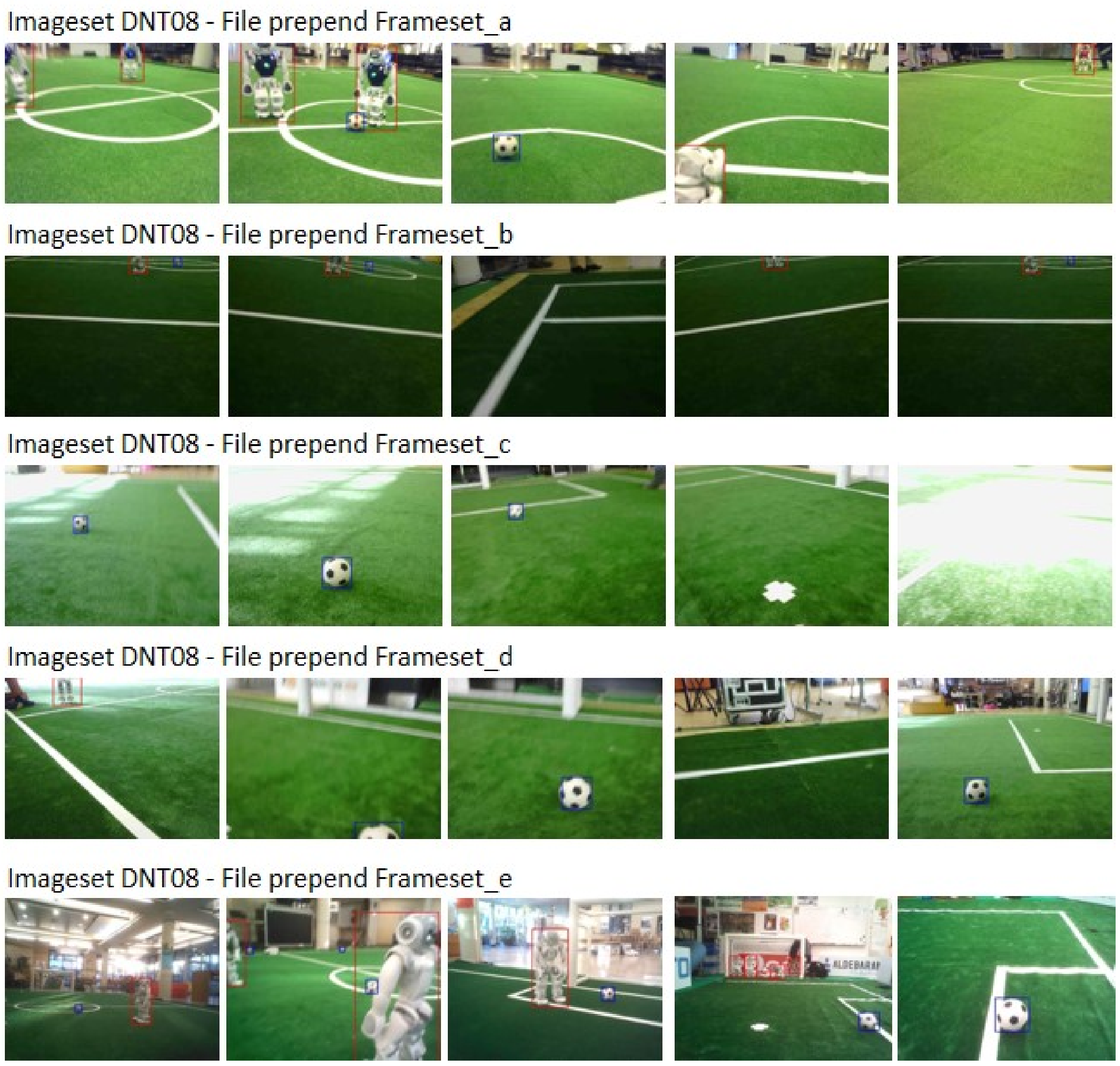
Table 3: Validation datasets, key statistics

C.2 Train Datasets - Image samples


Imageset 149 – File prepend 21_02_2018__17_02_52
Imageset 242 – File prepend 01_06_2018__17_17_01_up
Imageset 243 – File prepend 01_06_2018__17_17_01_low
Imageset 254 – File prepend 11_06_2018__16_35_19_low
Imageset 253 – File prepend 11_06_2018__16_35_19_up
Imageset 277 – File prepend 20_07_2018__15_14_36_up
Imageset 287 - File prepend 27_07_2018__15_44_10_up
Imageset 288 – File prepend 27_07_2018__15_59_59_lo
Figure 15: Sample images from training dataset with annotated ground truth bounding boxes 1/2


Imageset 289 – File prepend 27_07_2018__16_11_04
Imageset 290 – File prepend 27_07_2018__16_16_30
Imageset 309 – File prepend 22_08_2018__14_58_20_up
Imageset 310 – File prepend 22_08_2018__15_04_13_lo
Imageset DNT06 – File prepend frames_saves1
Imageset DNT07 – File prepend frames_saves2
Imageset DNT08 - File prepend frames_saves3
Figure 16: Sample images from training dataset with annotated ground truth bounding boxes 2/2

C.3 Validation Datasets - Image samples


Imageset DNT08 - File prepend Frameset_a
Imageset DNT08 - File prepend Frameset_b
Imageset DNT08 - File prepend Frameset_c
Imageset DNT08 - File prepend Frameset_d
Imageset DNT08 - File prepend Frameset_e
Figure 17: Sample images from validation dataset with annotated ground truth bounding boxes

D Evaluation of Yolo detection on Robots

Haar classifier - ball detection		
Precision	TP/(TP+FP)	83%
Recall / TPR	TP/(TP+FN)	19%
True Neg Rate TNR	TN/(TN+FP)	61%
Accuracy	(TP+TN)/TOTAL	23%
Balanced Accuracy	(TPR+TNR)/2	40%
F1	2TP/(2TP+FP+FN)	0,3

		Ground Truth: T	F	
Prediction:	T	81	17	98
	F	340	27	367
		421	44	465

Yolo - ball detection		
Precision	TP/(TP+FP)	100%
Recall	TP/(TP+FN)	70%
True Neg Rate TNR	TN/(TN+FP)	100%
Accuracy	(TP+TN)/TOTAL	72%
Balanced Accuracy	(TPR+TNR)/2	85%
F1	2TP/(2TP+FP+FN)	0,8

		Ground Truth T	F	
Prediction:	T	302	0	302
	F	131	32	163
		433	32	465

Yolo - robot detection		
Precision	TP/(TP+FP)	99%
Recall	TP/(TP+FN)	76%
True Neg Rate TNR	TN/(TN+FP)	91%
Accuracy	(TP+TN)/TOTAL	77%
Balanced Accuracy	(TPR+TNR)/2	84%
F1	2TP/(2TP+FP+FN)	0,9

		Ground Truth T	F	
Prediction:	T	336	2	338
	F	106	21	127
		442	23	465

Figure 18: Performance evaluation: Haar classifier (ball) vs. Tiny Yolo-v3/3L (ball and robot)

D.1 Ball Detection

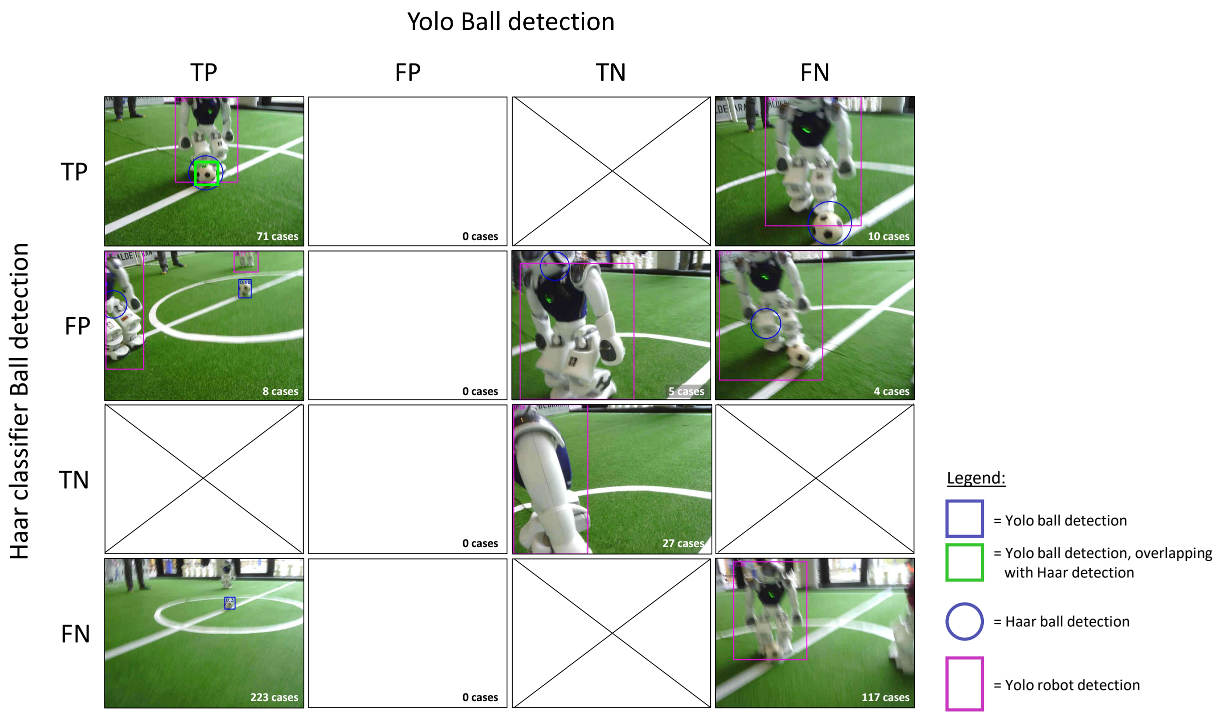


Figure 19: Illustration of all possible mixed classifier outcomes

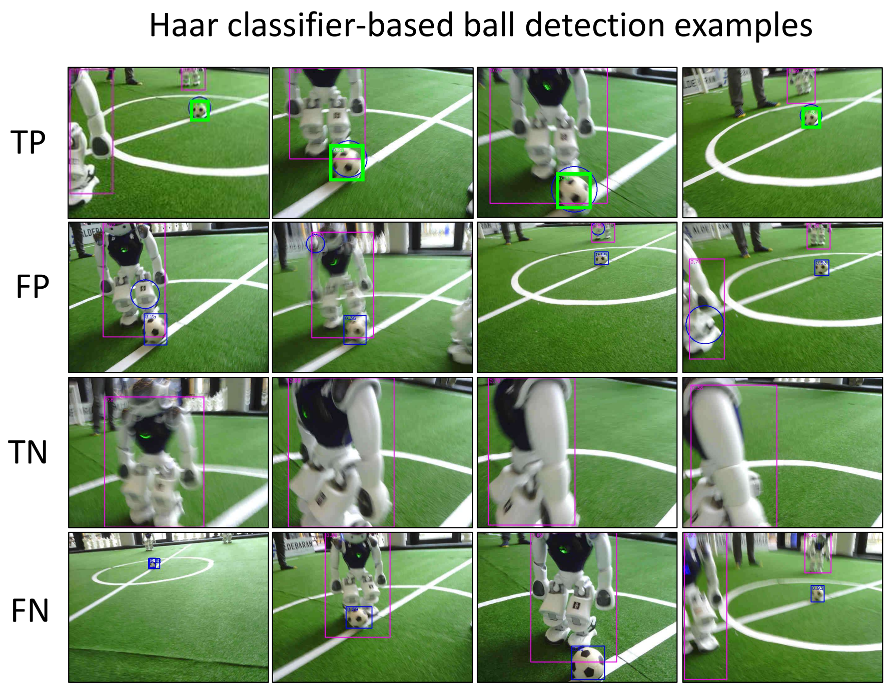


Figure 20: Illustration of Haar classifier-based detection outcomes

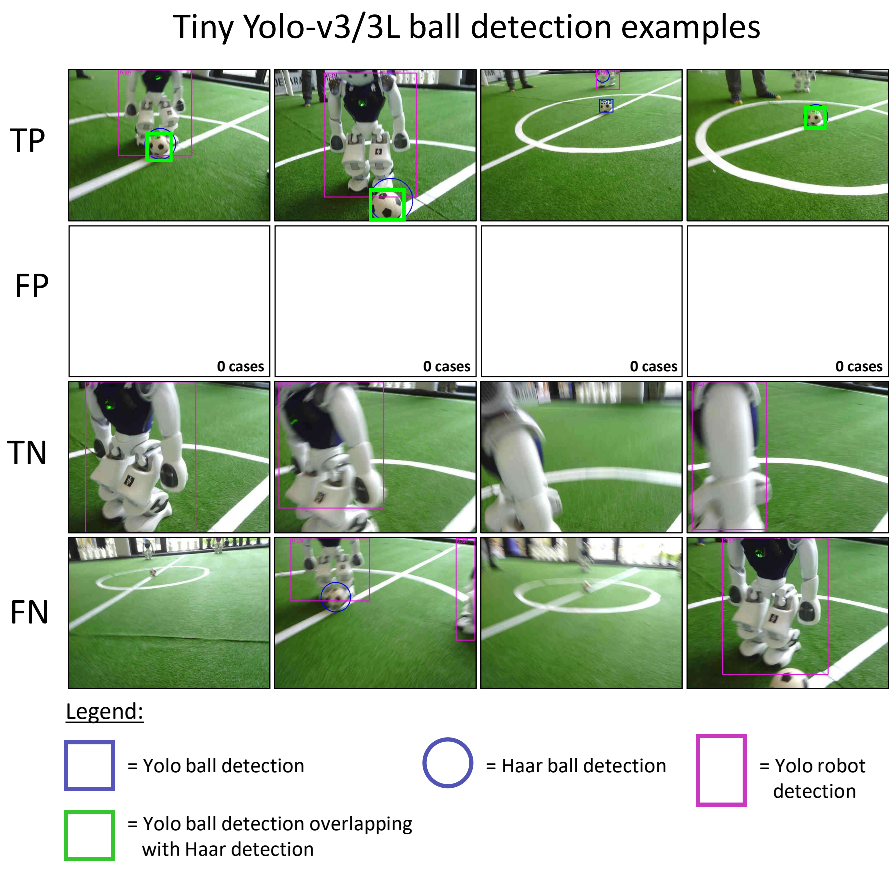


Figure 21: Illustration of Yolo outcomes for ball detection

D.2 Robot Detection

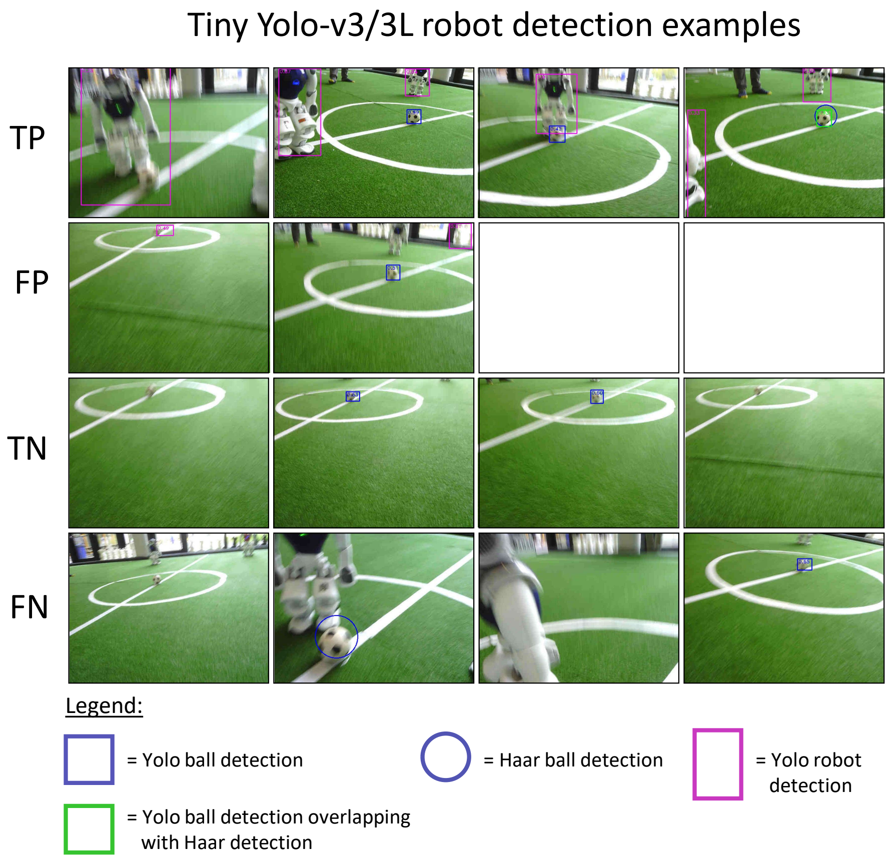

Tiny Yolo-v3/3L robot detection examples
TP
FP
TN
FN
Legend:
= Yolo ball detection
= Haar ball detection
= Yolo robot detection
= Yolo ball detection overlapping with Haar detection

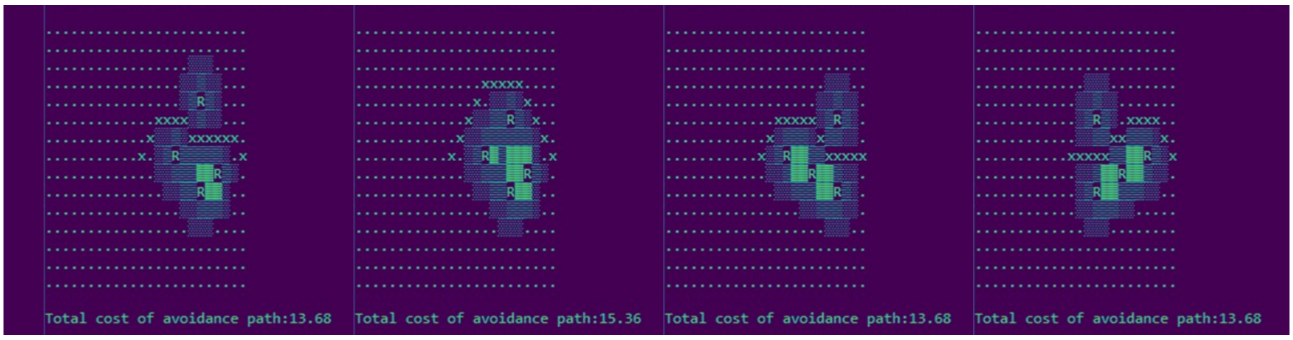
Figure 22: Illustration of Yolo outcomes for robot detection

E Navigation module tests

The navigation module uses a graph class for path finding:

- The field is divided into a graph of nodes and edges $G(N, E)$, with each node connected bi-directionally in 8 directions.
- An empty field will have edges with cost 1 (directions N,S,W,E) or $\sqrt{2}$ (directions NE,SE,SW,NW).
- The class implements a parameterized placement of obstacles: All edges originating from the obstacle node are assigned a peak cost c_{max}, which attenuates over adjacent nodes $\alpha^i \cdot c_{max}$, $i \in 1, ..n$ based on a the reach of the cost mountain n.
- This assignment is additive, meaning that if robots are near each other, their cost mountains will interfere and create a saddle. Hence, a 'cost landscape' is created.

E.1 Offline testing



```
Total cost of avoidance path:13.68
```

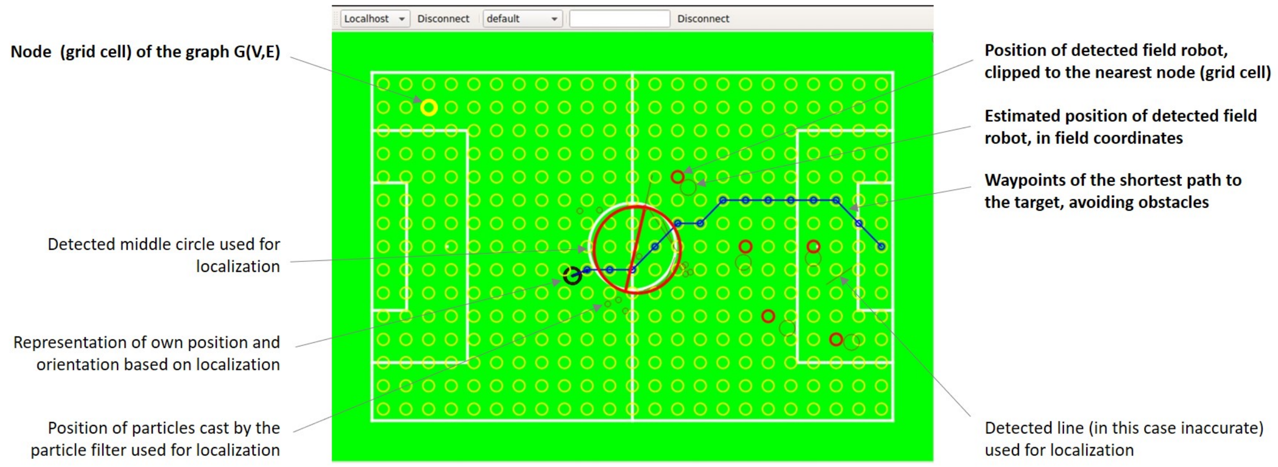
```
Total cost of avoidance path:15.36
```

```
Total cost of avoidance path:13.68
```

```
Total cost of avoidance path:13.68
```

Figure 23: Console output of offline evaluation of the path finding algorithm. Robot starts at center of the field and navigates to target goal on the right. R=obstacle, grid size (23×15), $c_{max} = 1000$, $\alpha = 0.75$, $n = 1$

E.2 Legend of the interface used for integration testing



Localhost
Disconnect
default
Disconnect
Node (grid cell) of the graph G(V,E)
Detected middle circle used for localization
Representation of own position and orientation based on localization
Position of particles cast by the particle filter used for localization
Position of detected field robot, clipped to the nearest node (grid cell)
Estimated position of detected field robot, in field coordinates
Waypoints of the shortest path to the target, avoiding obstacles
Detected line (in this case inaccurate) used for localization

Figure 24: Path finding implementation projected on the top field view of the DNT Interface. Example shown from a replay of a recorded integration test. Grid size (23×15), $c_{max} = 1000$, $\alpha = 0.75$, $n = 1$

F Background on techniques used by Yolo

F.1 Applying convolutions to sliding windows over an image

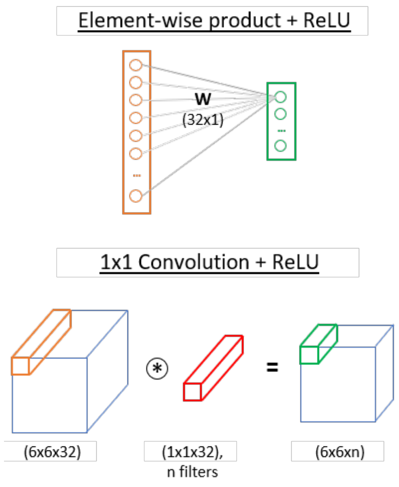


Figure 25: Substituting an element-wise product of a fully-connected layer with a (1x1) convolution[16]

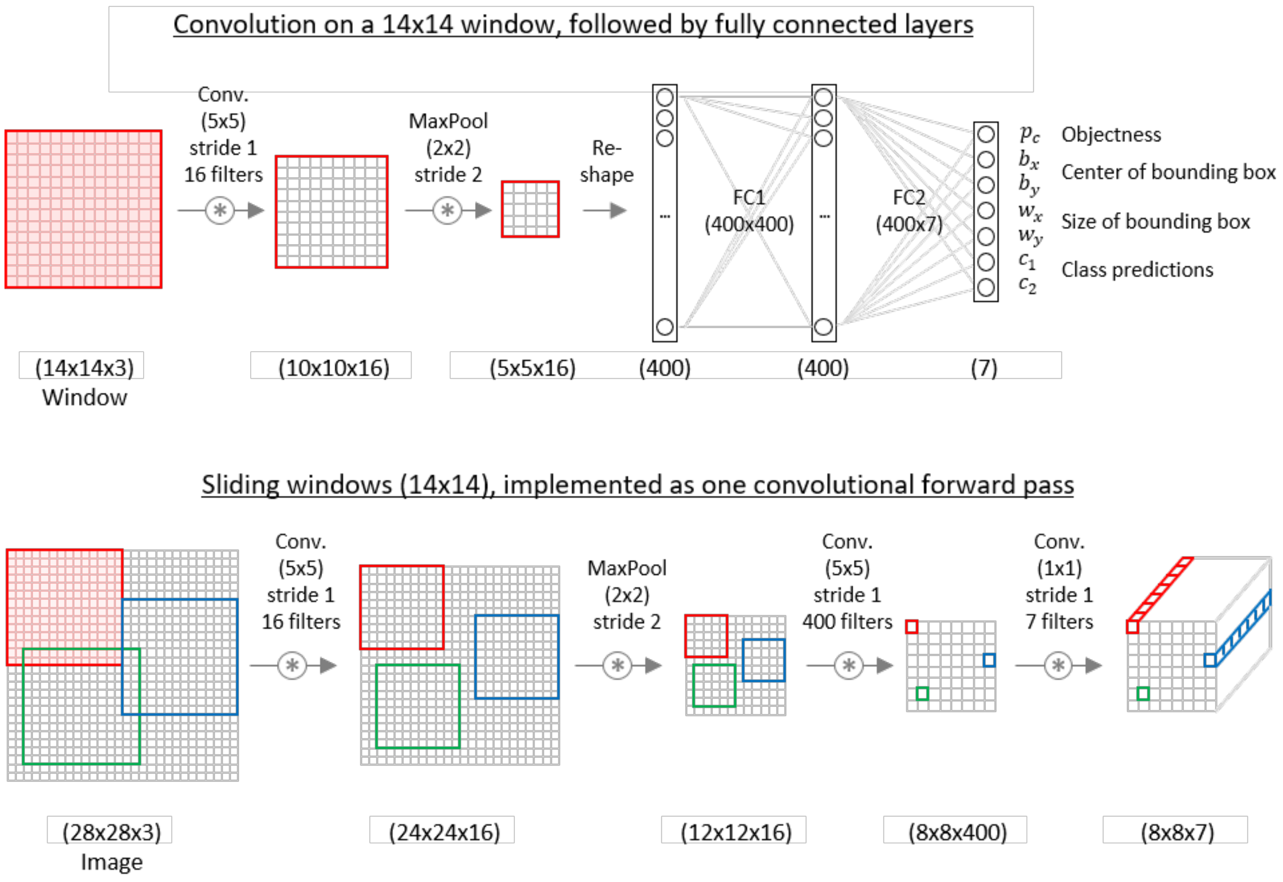
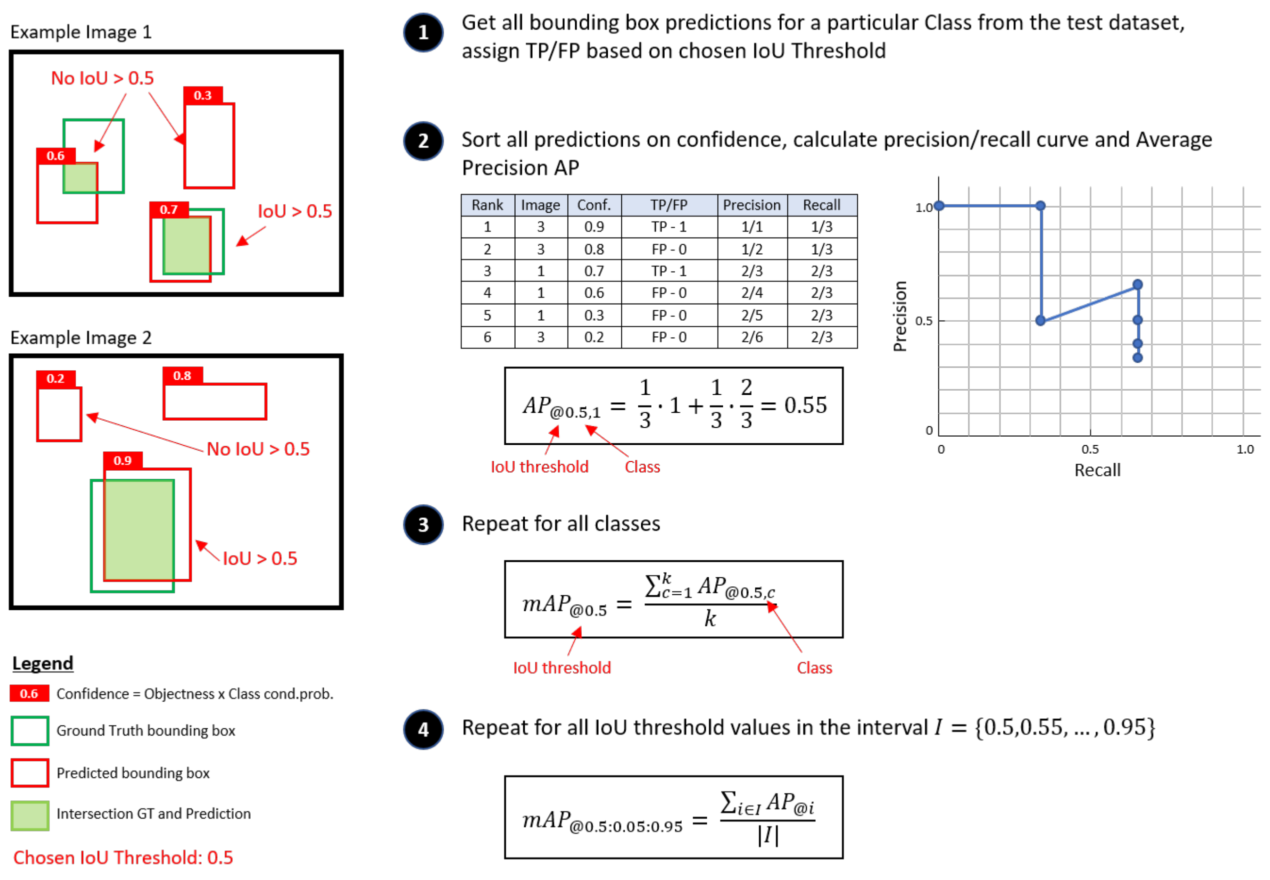

Convolution on a 14x14 window, followed by fully connected layers
Conv. (5x5) stride 1 16 filters
MaxPool (2x2) stride 2
Re-shape
FC1 (400x400)
FC2 (400x7)
p_c Objectness
b_x b_y Center of bounding box
w_x w_y Size of bounding box
c_1 c_2 Class predictions
(14x14x3) Window
(10x10x16)
(5x5x16)
(400)
(400)
(7)
Sliding windows (14x14), implemented as one convolutional forward pass
Conv. (5x5) stride 1 16 filters
MaxPool (2x2) stride 2
Conv. (5x5) stride 1 400 filters
Conv. (1x1) stride 1 7 filters
(28x28x3) Image
(24x24x16)
(12x12x16)
(8x8x400)
(8x8x7)

Figure 26: Illustration of the application of convolutions to sliding windows in a single forward pass using (1x1) kernels, as leveraged by Yolo (Image adapted from [18])

F.2 Evaluation using Mean Average Precision and IoU



1 Get all bounding box predictions for a particular Class from the test dataset, assign TP/FP based on chosen IoU Threshold

2 Sort all predictions on confidence, calculate precision/recall curve and Average Precision AP

Rank	Image	Conf.	TP/FP	Precision	Recall
1	3	0.9	TP - 1	1/1	1/3
2	3	0.8	FP - 0	1/2	1/3
3	1	0.7	TP - 1	2/3	2/3
4	1	0.6	FP - 0	2/4	2/3
5	1	0.3	FP - 0	2/5	2/3
6	3	0.2	FP - 0	2/6	2/3

$$AP_{@0.5,1} = \frac{1}{3} \cdot 1 + \frac{1}{3} \cdot \frac{2}{3} = 0.55$$

3 Repeat for all classes

$$mAP_{@0.5} = \frac{\sum_{c=1}^{k} AP_{@0.5,c}}{k}$$

4 Repeat for all IoU threshold values in the interval $I = \{0.5, 0.55, \ldots, 0.95\}$

$$mAP_{@0.5:0.05:0.95} = \frac{\sum_{i \in I} AP_{@i}}{|I|}$$

🟥 0.6	Confidence = Objectness x Class cond.prob.
🟩	Ground Truth bounding box
🟥	Predicted bounding box
🟩	Intersection GT and Prediction

Chosen IoU Threshold: 0.5

Figure 27: Workflow for evaluating mAP performance of a detection algorithm