REAL-TIME OBJECT DETECTION AND TRACKING

Andreas Grønbech Petersen s173901
Jonas Brøndum s173952
Philip Hoedt Karstensen s173941
Holger Christian Nyeland Ehlers s182521

ABSTRACT

Four Deep Learning models with three different backbone architectures were trained and applied to a simple object detection task. The trained models are used in combination with different tracking algorithms with the goal of reaching real-time object detection and tracking.

The first three models were developed from Faster R-CNN [1]. Two different backbones were used as pretrained models and were trained for 10 epochs. In general, the models have high precision, meaning that the estimated model parameters are able to predict a class fairly precisely. Reaching around 86% mAP. Conversely, the inference time varied a lot between the two backbones, with model 1 and 2 obtaining an inference time of 0.0664 s and 0.496 s while the third model attains one of 0.3238 s.

The fourth model uses the YOLO architecture which utilizes a single neural network that predicts bounding boxes and class probabilities directly from full images in one evaluation [2]. The model achieved the fastest inference time of all four models and an accuracy of 85.1%.

The three R-CNN models were also benchmarked on a video with several different objects. It was shown that the models perform better on a GPU. Additionally the smaller MobileNet-backbone performed the best with an average of 19.57 FPS rendition.

Different tracking algorithms have been explored: Centroid, KCF and SORT tracking. Through testing with different networks and scenarios it was concluded that the SORT tracking was the superior approach.

Index Terms— Deep Learning, Faster R-CNN, Single-Shot Detector, Object Detection, Object tracking, Real-time

Thanks to Peter Jensen for supervision and offering us the opportunity to work with this challenging and exciting project.

1. PROJECT DESCRIPTION

The purpose of the project is to develop a fast deep learning model for real-time detection of two objects as well as using the model predictions with a tracking algorithm.
The implementation of real world computer vision applications in industrial settings such as automated food production, rely on efficient AI algorithms as well as hardware solutions that can operate at scale. One compact solution is the use of a Nvidia Jetson Nano, however given its limited computational power, it requires sufficiently optimized AI algorithms.

2. METHODS

Multiple deep neural networks have been developed in the context of object detection models. This project implements two Faster R-CNNs (Region-based Convolutional Neural Networks) based on the two architectures ResNet50 & MobileNetV3, and the SSD (Single Shot Detector) YOLO (You Only Look Once) network based on the v5 architecture. This is an unofficial version of YOLOv5. The implementation of the three models are used for the detection of two classes of objects. The models are implemented in Python utilizing several libraries including the deep learning framework, PyTorch [3]. All code, data and auxiliary documentation is available at GitHub, *https://github.com/jonasbrondum/02456-project*.

2.1. Data set

The data set was created by collecting video recordings of beer (Grøn Tuborg) and cola (Coca Cola) cans rolling on the floor. The data set consists of 3389 frames from two video recordings. The videos were partitioned into frames that were manually labelled with boxes according to the presence of cans. The annotations were done using CVAT (Computer Vision Annotation Tool). The annotations were processed such that frames without annotations were disregarded to decrease subsequent processing time.
For good results the data needs to be pre-processed and used for training. Thereafter the results were post-processed for

easy interpretation of the performance and accuracy of the network (see also fig. 1).

2.2. Pre-processing

The 2 videos were split into separate frames and named accordingly. All frame partitions were annotated manually by the groups. Some frames were empty, while some contained several cans of different types. All frames were resized to 640x480 for faster training.

2.3. Training

The models were trained on a Tesla K80 using Google Colab. The Faster R-CNNs were trained using a generic modified script [4]. The YOLOv5 network was trained using a generic script from Roboflow [5].
For training and testing the entire data set was randomly shuffled and split with 80% for training and 20% for testing. The split was done using a seed.

2.4. Post-processing

Following the training each model was used for testing to obtain performance benchmarks. These benchmarks were completed to evaluate speed and precision. Furthermore, the detections were sent to the tracking section of the algorithm. The only post-processing performed was by the tracking algorithm.

3. NETWORKS

During the years several detection systems have been developed. This report focuses on two recent network approaches: R-CNN and YOLO.

3.1. Faster R-CNN

The Faster R-CNN is a state-of-the-art object detection network that relies on region proposal algorithms to hypothesize object locations [6]. Moreover, real time object detection models have to be fast and the Faster R-CNN achieves this by sharing the region proposal network's full-image convolutional features with the detection network. This enables nearly cost-free region proposals.
With this network approach two Faster R-CNNs were built; one with the MobileNetV3-Large backbone and one with the ResNet50 backbone. These backbones are used in the feature extraction step.

3.1.1. MobileNetV3

MobileNetV3 is defined as two models: MobileNetV3-Small and MobileNetV3-Large where this project uses MobileNetV3-Large [1]. MobilenetV3 is the third generation of the MobileNet and is built on the similar architectures of MobileNetV1 and MobileNetV2. The main contribution of MobileNetV3 is the use of the two Automated Machine Learning techniques MnasNet and NetAdapt for the identification of the best possible neural network architecture.

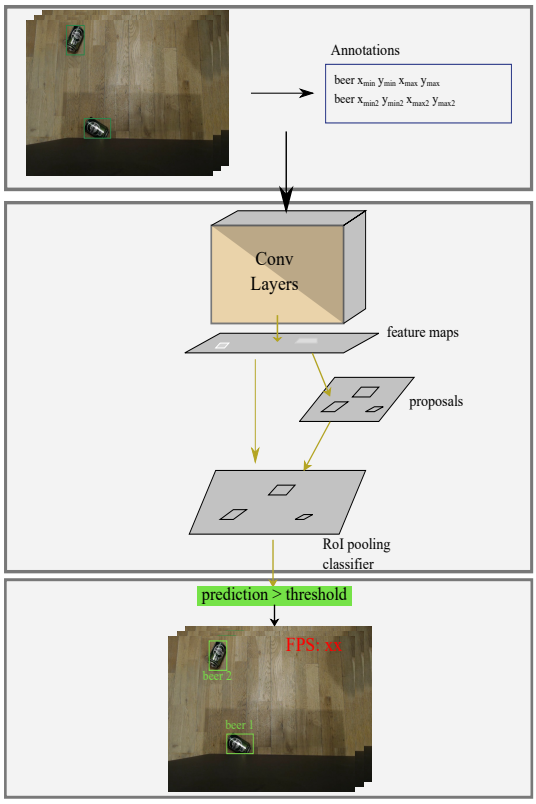

Fig. 1. The general algorithm from data collection to real-time tracking; here depicted with a Faster R-CNN.

3.1.2. ResNet50

ResNet50 is a residual network consisting of 50 layers. A general problem with deep networks is that as the depth of the network increases, the performance does not increase. This could be due to the vanishing gradient problem. ResNet50 uses a residual block layer that is an identity mapping from one layer to the next to prevent the vanishing gradient problem from happening. This allows for the training of deeper networks, as such it's also significantly larger than MobileNetV3.

3.2. SSD

SSD is an object detector framework that does not rely on region proposal networks. Hence, SSDs are very fast. Here, the SSD used (YOLO) unifies the separate components of object detection into a single neural network [2]. The YOLO detection system takes as input an image, resizes it, runs it through

a single convolutional network and finally thresholds the detections from the model's confidence level. Several versions of the YOLO network have been developed and this project uses YoloV5 which is a PyTorch extension of YOLOv3. The authors are aware of the controversy surrounding YOLOv5.

4. TRACKING ALGORITHMS

Object tracking is useful for keeping track of objects and their identities across frames. This is done by detecting the position of an object in a frame and then updating the position in the following frames. Even if the detection of an object fails, the tracking can still be maintained, because the previous frames still are considered. The ideal tracking algorithm will have the following traits:

- The ability to only require detection of an object once
- Fast performance
- The ability to handle objects that move out of the frame or disappear between frames
- The ability to take care of object occlusion

There are different types of algorithms that are able to perform object tracking and some are able to handle the individual traits mentioned above better than others. This means that there often is a trade-off when choosing a tracking algorithm. In this project several tracking algorithms have been investigated.

4.1. Centroid tracking

Centroid tracking is based on the Euclidean distance between two points. The centroid tracking algorithm receives bounding box coordinates from an object detector such as the Faster R-CNN. The initial centroids of the provided bounding boxes are calculated as the center coordinates of the bounding boxes and these are then assigned a unique ID. Object centroids are determined for every frame, but instead of assigning a new ID, the Euclidean distances between old and new object centroids are calculated. The smallest Euclidean distance is then used to associate old centroids with new centroids and the centroid coordinates are updated.
New objects that enter the frame will not be associated with any of the previous objects, since new object centroids that enter the frame will be registered as new objects. This also means that if the number of new centroids in a new frame is greater than the number of centroids in the previous frame, a new object ID will be registered.
The last step of the algorithm is to deregister old object centroids, which is done if the centroid cannot be matched to a previous centroid for a specified amount of frames.
There are two primary drawbacks of the centroid tracking algorithm. The first drawback is that every frame of the input video has to be passed through the object detector. This means that centroid tracking is not ideal with slow detectors that are computationally expensive, especially if the device using the detector is resource constrained. The second drawback is that occlusion poses an issue for the algorithm. This is mainly seen when objects get too close to each other, which can lead to that the IDs of the centroids are being switched around. The centroid tracking algorithm used in this project is based on the algorithm presented by Adrian Rosebrock on PyImageSearch [7].

4.2. Kernelized Correlation Filter

Kernelized Correlation Filter (KCF) is a tracking method originally proposed by Henriques et. al. [8]. The implementation used for this project is extended to include color-features as described in "Adaptive Color Attributes for Real-Time Visual Tracking" by Danelljan et. al. [9]. This is a real-time tracking method which utilizes luminance and color features together with the kernelized least squares classifier.
Briefly described, the algorithm works in the following way: the tracker is initialised using an image and a target image patch. From this, the tracker considers all cyclic shifts in the patch as training data, `x`, which using a Gaussian function is mapped to an output, `y`, such that input-output pairs exists. A mapping to a nonlinear space is performed using a Gaussian RBF kernel. A classifier is then trained by minimizing a least squares cost function of the difference between the `x` mapped to the nonlinear space and `y`. There are many more details to the algorithm, which can be found in the original papers.
KCF tracking should be able to perform well with colorful objects and promises to run in real-time. The implementation found in the OpenCV library for Python will be used.

4.3. SORT

SORT (Simple Online Real-time Tracking) is a tracking algorithm which seeks to be a simple and effective alternative to solving the multi-object tracking problem, specifically for online and other real-time applications [10]. Since this method is based on a Kalman filter, a model of the tracked object is needed. The model is used to predict the movement between frames. The object can be modelled by its linear velocity, where the states are pixel positions and scale, their derivatives with respect to time and the aspect ratio of the bounding box. However, the aspect ratio is considered constant.
The velocities of the object are solved for by the Kalman Filter framework. The bounding box from the detection network is assigned to a target in the frame by having each target's bounding box geometry estimated by the filter, then computing the IOU distance between detections. The actual assignment is solved by use of the Hungarian Algorithm. The covariance of the velocity of a new object is set very high to reflect the uncertainty. This method promises to deliver very fast and robust tracking which also can handle short term occlusions.

5. IMPLEMENTATION

To combine the entire solution, data source, neural network and tracking, Python was used. The video source, either a prerecorded video or live video feed from i.e. a webcam is passed using OpenCV. The network model is loaded as a standard PyTorch model, and no significant optimization is performed. Results of the tracked objects are shown using OpenCV as the predicted bounding boxes on top of detections in the image.
Every implementation seen here will follow roughly the same structure: Load in PyTorch model, and set to evaluation mode. Configure video source and initialise tracker. Then perform the following until there are no more frames: Load next frame, pass it through the network to get predictions, pass predictions to tracker, show result.
When evaluating the performance of the model, both GPU and CPU acceleration will be considered.

6. RESULTS

6.1. Train and test results

The train loss and accuracy for the different R-CNN models can be seen in fig. 2. The train loss and accuracy for the SSD-network, YOLO, can be seen in fig. 3.

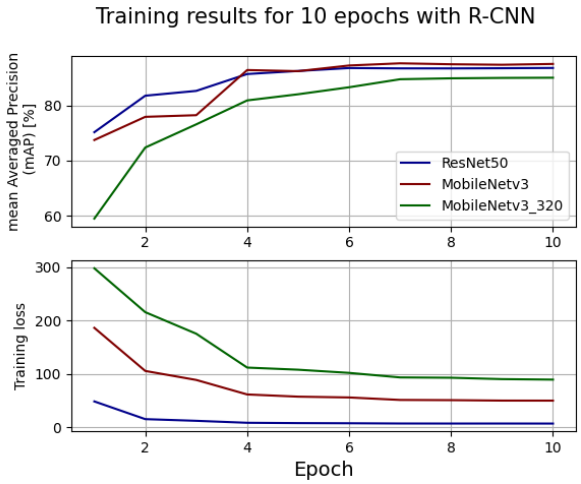

Fig. 2. Training loss and precision over 10 epochs of training for the three Faster R-CNNs.

The ResNet50, MobileNetv3 and MobileNetv3_320 were all trained for 10 epochs to get a perspective of their convergence to the data and their performance increase. The YOLO-network was trained for 100 epochs with similar convergence results. It's seen that the YOLO-network through the training is unstable in its precision.

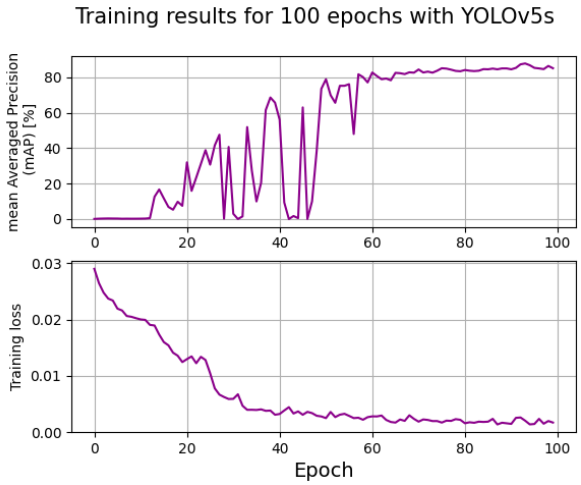

Fig. 3. Training loss and precison over 100 epochs of training for the SSD-network; YOLOv5s.

What is common for all the four models is that they reach a saturation in regards to both losses and precision around 2/3 of the training, which could suggest that the data set is limiting and the training could be stopped early, since the accuracy has saturated.
For an R-CNN or SSD network the evaluation is focused on the predictive precision of bounding boxes, also referred to as mAP (mean Averaged Precision) [6]. For this project inference time, the time to do one forward pass in the network, is crucial for the second section, real-time tracking.
The test results can be seen in table 1 and figure 4.

Table 1. Accuracy and inference time achieved by pre-trained CNN models.

Backbone	Accuracy (mAP)	Inference Time [s]
MobileNetv3Large	87.5%	0.0664
MobileNetv3Large_320	85.0%	0.0496
ResNet50	86.8%	0.3238
YOLOv5s	85.1%	0.0333

From the results it's clearly seen that all of the networks obtain very similar accuracies. However the three networks: MobileNetv3, MobileNetv3_320 and YOLOv5s are all significantly faster than ResNet50 with MobileNetv3 seemingly compromising better between speed and accuracy.

6.2. Model live performance

In order to further evaluate the algorithm, an additional video was recorded with different types of cans including, Grøn Tuborg, Coca Cola et. al. It was used to assess the speed and thus computational demand of each model.
The models were evaluated on the entire video of 43 seconds

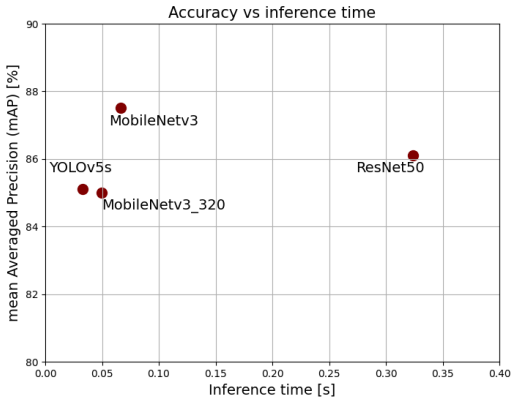

Fig. 4. Accuracy vs inference time for the different trained models.

on both an Intel i5-4460K CPU and an Nvidia GTX 1050Ti GPU. The results can be seen in table 2. Unfortunately this test wasn't successfully completed with the YOLO-network. The reader is referred to the GitHub repository for the video.

Table 2. Processing performance, both averaged and interval, of the different models using either CPU or GPU. The processing is performed on a separate test video with various types of cans: Grøn Tuborg, Coca Cola, Heineken, Royal Export, Royal Export Jul and Fanta. $[FPS]$ = minimal and maximum FPS captured, $\overline{FPS}$ = average FPS for entire video and t = elapsed time for tracking of entire video.

Model	$\overline{FPS_{CPU}}$	$[FPS_{CPU}]$	t[s]
MobileNetv3Large	1.34	0.76 - 1.55	933.25
MobileNetv3Large_320	5.5	1.63 - 7.08	228.16
ResNet50	0.23	0.12 - 0.24	5416.15
Model	$\overline{FPS_{GPU}}$	$[FPS_{GPU}]$	t[s]
MobileNetv3Large	9.88	0.39 - 14.21	126.95
MobileNetv3Large_320	19.57	0.39 - 29.59	64.08
ResNet50	2.17	0.05 - 2.65	577.29

From the results it's clear that there is a significant difference between the computational performance between the different models.
Additionally it's clearly seen that the speed of a GPU for tasks of this type considerably outperforms a CPU. In terms of the models tested, here the MobileNetv3Large_320 with the GPU as seen from the table averages an FPS of 19.57, which is, with a "real-time" being 30 FPS, 2/3 "real-time". This also aligns with the expectations given the inference time in table 1.
Furthermore, from the test it was experienced that while the ResNet50-network was slower on performance it generally detects cans better with a higher confidence. The faster performing models had some issues with detection in high-speed movement and with unfamiliar colors (eg. Royal Export Jul, which is a blue can). One thing to note is that this performance test was completed with visual rendition of the video through OpenCV, meaning that the final results could have been slightly faster if not rendered live.

6.3. Robustness of tracking

The centroid tracking performed satisfactory. The algorithm was able to run near real time when used in combination with a fast detector. The main issue that was observed was object ID switching, which occurred when cans collided or came close to each other.
The performance of the KCF tracking was unsatisfactory, so it was not used further in the project. It ran in real time, but had trouble tracking multiple cans, likely because cans of the same class have the same color.
SORT ran in real time with no significant computational resources required. Compared to the two other tracking algorithms, SORT is superior in every way, but especially when considering cans colliding. None of the algorithms work particularly well with occlusions, but SORT can be configured to handle occlusions better if desired, by increasing the amount of frames before an object is discarded, but this can increase the amount of old tracklets in an image. This is a trade-off that must be made on a case-by-case basis. A few images showing the different scenarios of cans colliding using either SORT or centroid tracking can be seen in the plots folder in the GitHub repository.

7. DISCUSSION

We used YOLOv5 to obtain faster inference time as it is much faster than the Faster R-CNN models. This is due to the single shot design where SSD methods integrate the region proposal network and the classification localization computation. Faster R-CNN uses regional proposal network where two shots are needed - one for generating region proposals and one for detecting the object of each proposal - thus, SSD approaches such as YOLOv5 are much faster. Additionally, SSD models can detect any amount of objects while the computational load is invariant as it is only based on the number of anchors. For Faster R-CNN the computational load scales with number of regions proposed. A drawback with SSD models is that they in general are less accurate whereas Faster R-CNNs are slower and takes longer time to train.

It was not possible to do a direct comparison of the Faster R-CNN models and YOLOv5 using the newly recorded video. Hence, the robustness and inference time of YOLOv5 on a similar video is unknown.

As mentioned in section 2, the data set was sampled randomly with a 80/20 split. Furthermore, it was seen from the results that all the network architectures achieved very high and similar precision for the test data. This could be due to the fact that the training and test data is inherently very comparable. An example of this could be where frame 2 and 4 are training data and frame 3 is test. Realistically there isn't a large change in the scene between these three frames. This ultimately means that the way the data set has been collected and sampled can lead to over-fitting in the training, which in turn leads to an over-fit in the test data.
For closed environments, such as the ones the project stem from, this is not necessarily a negative feature since the amount of different scenarios are limited. However it also means that the results regarding average precision for testing become less relevant without further evaluation.

8. CONCLUSION

Different detection networks have been evaluated and tested with the generated custom data set. The networks have been trained using this data set. Different tracking algorithms have been investigated and implemented as well. In conclusion, a solution which can run in near real-time (19.5 fps) has been achieved, which also tracks all relevant objects in-frame satisfactory.

9. FUTURE WORK

For the project several optimization options were abandoned due to lack of time, however as seen from the results, it's possible to improve on the algorithm and future results going forward. This includes network efficiency that could have been improved by various complementary efforts for instance quantization that reduces precision arithmetic [1], thus increasing the speed of inference for both CPU and GPU.

To improve the accuracy and robustness of the models it would definitely be advantageous for future iterations to perform image augmentation. Several types of augmentation could be considered:

- Random Cropping
- Rotation
- Random deletion of image sections
- Randomized color skew

Since it's slightly more complex to do image distortion of any kind and keep the target labels consistent with the newly augmented images, these represent more cumbersome implementations. However, a more promising approach could be color skewing, with randomized changes in brightness, contrast, saturation and hue. This could potentially "double" the data set without having to change the target labels.

Furthermore, to truly evaluate the solutions attained, they must be run on real-time GPU hardware such as the Jetson Nano. This would likely also require significant effort, and thus have not been carried out yet.

10. REFERENCES

[1] Grace Chu Andrew Howard, Mark Sandler et. al, "Searching for mobilenetv3," 5 2019.

[2] Ross Girshick Joseph Redmon, Santosh Divvala et. al, "You only look once: Unified, real-time object detection," 2015.

[3] Francisco Massa Adam Paszke, Sam Gross et. al, "Pytorch: An imperative style, high-performance deep learning library," 2019.

[4] Several contributors, "Torchvision object detection finetuning tutorial," .

[5] Jacob Solawetz and Joseph Nelson, "How to train yolov5 on a custom dataset," .

[6] Shaoqing Ren, Kaiming He, and Ross B. Girshick et al., "Faster R-CNN: towards real-time object detection with region proposal networks," *CoRR*, vol. abs/1506.01497, 2015.

[7] Adrian Rosebrock, "Simple object tracking with opencv," Jul 2021.

[8] João F. Henriques, Rui Caseiro, Pedro Martins, and Jorge Batista, "Exploiting the circulant structure of tracking-by-detection with kernels," in *Computer Vision – ECCV 2012*, Berlin, Heidelberg, 2012, pp. 702–715, Springer Berlin Heidelberg.

[9] Martin Danelljan, Fahad Shahbaz Khan, Michael Felsberg, and Joost van de Weijer, "Adaptive color attributes for real-time visual tracking," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2014.

[10] Alex Bewley, Zongyuan Ge, Lionel Ott, Fabio Ramos, and Ben Upcroft, "Simple online and realtime tracking," *2016 IEEE International Conference on Image Processing (ICIP)*, Sep 2016.