Embedded Real-Time Object Detection for a UAV Warning System

Nils Tijtgat[1], Wiebe Van Ranst[2], Bruno Volckaert[1], Toon Goedemé[2] and Filip De Turck[1]

[1]Universiteit Gent
Technologiepark-Zwijnaarde 15, 9052 Gent, Belgium
nils.tijtgat@ugent.be, bruno.volckaert@ugent.be, filip.deturck@ugent.be

[2]KU Leuven
Technologiecampus DE NAYER, KU Leuven, Jan de Nayerlaan 5, 2860 Sint-Katelijne-Waver, Belgium
wiebe.vanranst@kuleuven.be, toon.goedeme@kuleuven.be

Abstract

In this paper, we demonstrate and evaluate a method to perform real-time object detection on-board a UAV *using the state of the art* YOLO*v2 object detection algorithm running on an* NVIDIA *Jetson* TX2*, an* GPU *platform targeted at power constrained mobile applications that use neural networks under the hood. This, as a result of comparing several cutting edge object detection algorithms. Multiple evaluations we present provide insights that help choose the optimal object detection configuration given certain frame rate and detection accuracy requirements. We propose how this setup running on-board a* UAV *can be used to process a video feed during emergencies in real-time, and feed a decision support warning system using the generated detections.*

1. Introduction

Unmanned Aerial Vehicles (UAVs) have recently evolved from nice to have gadgets to full-fledged industrial grade workhorses in many domains. Precision agriculture [2] [23], industrial inspection [15] and package delivery service of the future [8] are only a few examples of the many domains where UAVs can prove their worth. UAV sales have as such immensely increased these past few years, and reports indicate this growth will continue well into the second decade of this century [7]. This fast growing market has sparked investments and advancements in UAV technology, and economy of scale has brought the cost down. Recent consumer devices display an impressive set of specifications and features at an affordable price. Industrial UAVs are meanwhile equipped with high-resolution sensor arrays to perform a very diverse set of applications, and are rugged enough for all-weather outdoor operation. Parallel to UAV technology advancing, small form factor computing boards for embedded applications have become integrated powerhouses that enable edge computing on mobile devices. Running convolutional neural networks on embedded systems has become a reality. Combining the technological novelties from both the UAV and embedded computing domain allows for some very interesting new approaches, which we will apply in this paper from an emergency operation management perspective. UAVs can provide invaluable visual information during incidents and help rescue workers and coordinators better tackle the situation at hand. Aerial imagery can indicate possible entry-points during a fire, help locate missing persons during a search & rescue operation, locate possible dangerous containers on-site, etc. This is the issue we will address in this paper: how can we perform accurate real-time object detection on-board a UAV, and integrate that into an emergency situation warning system? We specifically choose UAV on-board object detection, as a robust wireless link during an emergency is not always guaranteed, or the link lacks the bandwidth or availability to reliably support a video stream. As features and application domain evolve at high pace, the major current UAV technology Achilles' heel remains the limited flight time. Even though lithium-ion battery technology is rapidly advancing, future technological improvements promise evolution rather than revolution when it comes to maximum energy capacity. Peripheral hardware weight and power consumption on-board our UAV should therefore always be carefully considered and kept as low as possible, limiting the available performance for our object detection algorithm.

2. Approach

The context in which we report the findings of this paper, is that of a UAV platform capable of arriving at an emergency scene on very short notice to provide increased situational awareness for emergency response coordination. The UAV could be stationed on top of the fire station or a regional launch site and depart as soon as a distress call rolls in. Using a combination of a 'straight as the crow flies'

path and predefined paths at specific heights to avoid known obstacles in the area, the UAV can ignore traffic and autonomously arrive on scene faster than emergency responders can. Once on-site, it helps decision makers assess the situation more efficiently and coordinate the operation with greater ease as would normally be the case. The UAV should have a high degree of autonomy and accept high level commands to perform in-depth inspections of the area (including e.g. capturing additional imagery of heavily impacted areas, monitoring operational progress). An on-board high resolution optical camera offers decision makers a thorough view on the situation. To assist operators even more during stressful situations (where information overload can occur at any given time), an automated decision support system notifies of ongoing relevant events. As an example: if a rescue worker approaches a dangerous situation (high pressure gas pipe, burning gas tank) a warning is generated to alert those involved of the potentially imminent danger. One vital module of this decision support system, is the object detection algorithm that looks for predefined items and passes this detection on to the decision engine. In this engine, the location of the detected object is estimated so that reasoning can take place in correlation with different objects present (distance of a fire truck to a burning barrel for instance). This paper focuses on the on-board object detection algorithm and integration with risk assessment. People, emergency responders, vehicles, dangerous goods containers, gas tanks, are some examples of the predefined objects we want our model to detect.

3. Related work

3.1. Object Detection

Traditionally, before the rise of convolutional neural networks (CNN), object detection (and by extension person detection), was done using hand crafted features. The Viola and Jones [27] face detector proved that is was possible to get really good performance for face detection using integral images in combination with an AdaBoost classifier. Other detectors like Histograms of Oriented Gradients (HOG) [3], Internal Channel Features (ICF) [5], Aggregated Channel Features (ACF) [4] and Deformable Parts Model (DPM) [6] make use of histograms of oriented gradients which have proven to get really good results for pedestrian detection and object detection in general. These detectors use features that are engineered and use classical machine learning approaches like decision trees / AdaBoost (ACF, ICF, V&J) or support vector machines (SVMs) (HOG, DPM). They generally use a sliding window approach, which evaluates a fixed size window at all positions in the image at different scales to get detection results for the entire image.

Many of these methods can easily run in real-time on embedded platforms as was demonstrated by [9] for use on drones. For some time now however, deep learning methods offer better accuracy at the cost of speed. Until recently it was not feasible to get them to run real-time on embedded platforms as the neural nets they are built on require a lot of GPU computational power. Recent developments have improved this, as deep learning methods are now catching up in terms of speed. In what follows, we will analyze the ac-

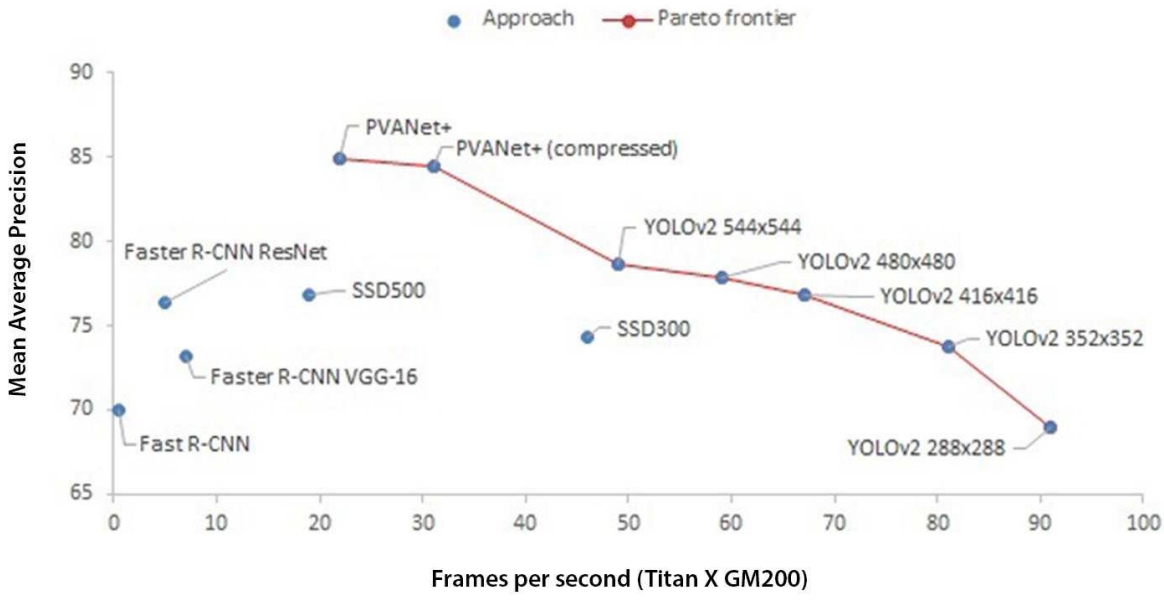

Figure 1. Comparison of different object detection algorithms tested on Pascal VOC2007 data (trained on VOC2007 and VOC2012). Image courteously provided by AlexeyAB from his darknet GitHub repository (https://github.com/AlexeyAB/darknet).

curacy/speed trade-off of both of these methods. We compare the ACF hand crafted features detector which is still close to state of the art, to the YOLOv2 and TinyYOLO detector that offer a good trade-off in terms of accuracy/speed on the deep learning side.

Deep convolutional neural networks form the current cutting edge technology when it comes to object detection. Recent literature provides ample examples of CNNs being used for a wide range of detection tasks [12, 22, 24, 14]. Only a selection of the proposed approaches qualifies as a real-time algorithm, a crucial requirement for our application. These are mostly single stage methods, as a two stage approach is computationally expensive. A first single stage approach is SSD [14]. It is surpassed in most of its configurations however by YOLOv2 [20] proposed by Redmon & Farhadi in both detection speed and detection accuracy. YOLOv2 is the successor of YOLO [19] and runs on the open source C Darknet neural net [18] by the same authors. A very interesting extra model running on this same underlying Darknet neural net is the TinyYOLO detector. It has a slightly lower detection accuracy, but the attainable frame rate is 4 times higher than YOLOv2 can achieve[1]. Single stage detectors often struggle to keep detection accuracy up to par when compared with two stage approaches, but Ren et al. [21] prove recent advances can even outperform the latter.

Looking at other CNNs that were designed with an embedded application in mind, we find LCDet [26] among others. Inspired by the YOLO architecture, LCDet presents a TensorFlow [1] implementation focused on keeping the neural net small: it features a 4-fold reduced memory usage and improved performance with only a minor drop in detection accuracy. The paper presents a benchmark where LCDet proves to be more accurate than the first generation YOLO, but the authors state YOLOv2 should empirically be more accurate than LCDet. Another approach to greatly reduce the CNN footprint is SqueezeDet [28], mainly focusing on autonomous driving. A final proposal by Kim et al. to implement lightweight neural networks for real-time object detection is PVANET+ [11]. Figure 1 is a graphical comparison of most of the previously mentioned models that plots the frame rate versus the mean average precision. The frame rates indicated are attained using a powerful Titan X (GM200) GPU and far greater than we will be able to attain, but still provide a great comparison between the algorithms.

3.2. Embedded platform

When it comes to choosing a UAV on-board embedded system for image detection, Hulens et al. [10] present a survey of different platforms with their respective computing power and influence on system battery life. The paper mentions the Jetson TK1 (predecessor of our Jetson TX2) and rates its CPU performance below average. This makes perfect sense: the Jetson is a powerful GPU embedded platform and the benchmarking algorithm used is CPU-based. The main algorithm we will be running is GPU based. The paper hence does not directly show us what GPU platform to choose, but the ideas and effects on battery lifetime of different embedded platforms remain valid and relevant.

3.3. Object positioning algorithm

Once our embedded system has detected objects of interest, we determine their position to feed this information to the decision support system. Preferably we want a solution that doesn't require a complex on-board setup (LiDAR or other expensive and/or heavy imaging system) to minimize cost and maximize flight time. Tijtgat et al. [25] present a solution that matches this requirement and provides a positional accuracy of $<$1.5m. The work presented in [13] proposes a vision-based UAV approach and landing scenario. This could serve as inspiration, with the limiting factor that the UAV would have to hover over the detected object's location to determine its position.

4. Architecture

The current architecture is the result of a set of design decisions upon which we will elaborate in the following subsections. As an example, Figure 2 illustrates what the architecture looks like for our use case. An autonomous UAV captures video data that the on-board hardware processes. In the example, a 'Firefighter' and two 'Barrel' instances are detected and their position is calculated. This information is relayed to the decision support system, that generates an alert if the firefighter gets too close to the dangerous products.

4.1. Object detection algorithm

The object detection algorithm we choose will have a major influence on the performance and battery life of our application. Many recent implementations are available, each having their merit. We compare the YOLOv2 [20], TinyYOLO and ACF [4] algorithms because of their excellent balance between frame rate and detection accuracy. Both YOLO algorithms run in the same Darknet framework, which allows us to easily run two different configurations and compare results. For ACF we use our own C++ implementation and also evaluate an experimental GPU port.

4.2. GPU implementation of ACF

In order to evaluate the performance of the ACF detector on an embedded GPU platform like the Jetson TX2, we also evaluate an embedded GPU implementation of the ACF detector. Porting the detector to GPU consists of two different stages: first we calculate the ACF features, after which we evaluate these features using a decision tree that was learned by the AdaBoost algorithm. Fast real-time performance is a requirement, so only evaluation time is important. Training is kept on the CPU (apart from hard negatives mining).

[1] https://pjreddie.com/darknet/yolo/

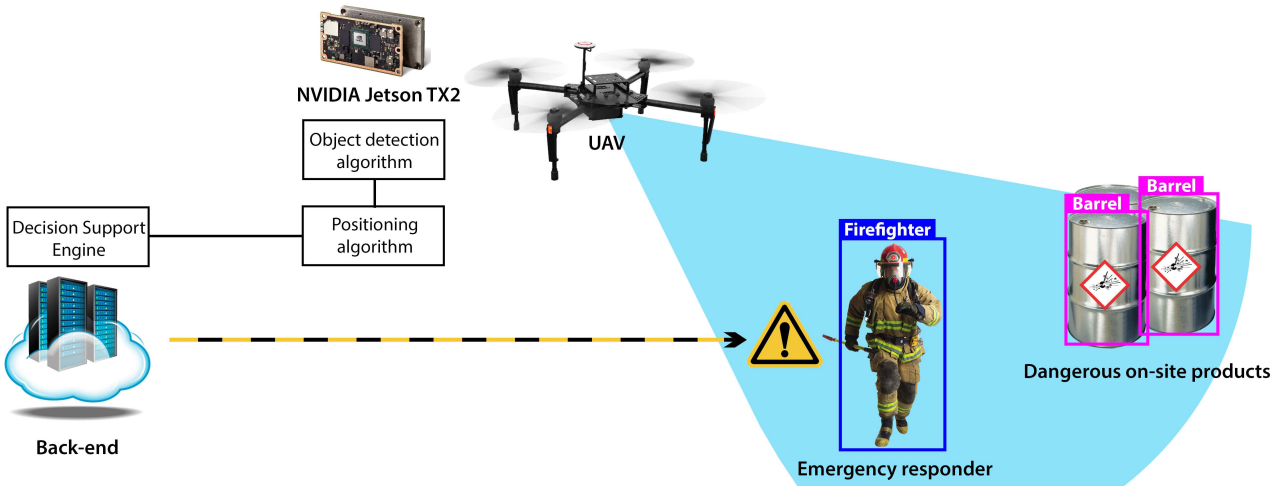

Figure 2. System architecture of a UAV-based decision support system for emergency response

To calculate the different ACF features on GPU we use the NVIDIA performance primitives (NPP) library that contains many standard image processing functions like color conversion, gradient calculation, rescaling etc. In fact, most of the operations used to calculate ACF features can be found in this library. To calculate the HoG from the gradient for histogram binning, we created our own CUDA implementation.

We tried many different approaches for the evaluation. An obvious place where parallelism can be found in the ACF algorithm is in the sliding window stage. Many windows (one window for every position in the image) have to be evaluated. As there are no data dependencies between windows, this can easily be done in parallel. We call this approach the window parallel approach.

Another place where we find parallelism in the algorithm, is in the evaluation of different decision trees for each window. ACF uses a series of weak classifier to create one strong classifier that outputs a score that is the sum of all the previous weak classifiers. The big advantage of this scheme is that it is not required to evaluate every classifier for each window. From the moment we determine that the first weak classifiers return a weak score for a window, we can rule out the window and stop evaluation early. This speeds up the evaluation tremendously. On GPU however, stopping early with the evaluation of decision trees creates control divergence, and reading many sparsely populated values from memory also gives disappointing memory performance on an already memory bound problem.

To get rid of the control divergence we can also evaluate decision trees in parallel. We call this the stage parallel approach, as we evaluate weak classifiers in parallel. We take groups of k (for instance $k = 128$) decision trees, evaluate them in parallel, calculate the global score (sum of all k classifiers) and if we are still above the predetermined threshold launch the next batch of k classifiers.

In the end we choose a hybrid implementation inspired by [17] (for Viola and Jones classifiers) that uses both of these approaches. We first evaluate each window in the image in a window parallel approach, after which we use dynamic parallelism to launch additional kernels that take a stage parallel approach. The results of our GPU implementation can be found in section 5.3.

4.3. Embedded GPU processing platform

Since YOLOv2 heavily relies on the CUDA toolkit, we need to find a powerful GPU platform that is both constrained in weight, dimensions and power consumption. Modern desktop GPUs require big amounts of power to run under full load (NVIDIA GTX 1080Ti has a TDP of 250W) but deliver high amounts of floating point operations per second (TFLOPS) in exchange (11.3TFLOPS in the example of the GTX 1080Ti). While the raw power of this type of GPU is perfect during the offline and unconstrained training phase, it is clear that we need a completely different approach to meet our requirements (minimal power consumption, limited onboard processing power, minimal weight) for the inference phase on-board the UAV.

To determine if weight or power consumption should be the key parameter to consider when choosing our embedded platform, let's have a look at equation (1)[2]. This equation describes the power P (in Watts) required to hover an aircraft of mass m (in kg) with a propeller of radius r (in meters). K depends on the air density Q_{air} as defined in equation (2) and g is the gravitational acceleration with a conventional standard value of exactly 9.80665 m/s^2. At 20°C and a pressure of 1atm, K has a value of 0.363562254.

$$P = K \cdot \frac{(m \cdot g)^{3/2}}{r} \tag{1}$$

$$K = \sqrt{\frac{1}{2\pi \cdot Q_{air}}} \tag{2}$$

[2] http://www.starlino.com/power2thrust.html

As an example, a commercially readily-available DJI Phantom 4 weighs in at 1380g total takeoff weight and has a propeller radius of 12cm. According to equation (1), it requires just over 150.83W of power divided over its four propellers to hover. This theoretical result does not take into account the power required to operate the flight controller and other on-board electronics and sensors. It's clear that the $m^{3/2}$ factor in (1) progressively penalizes every bit of added weight. Adding 100g of weight (+7.2%) to previous example increases the power required by 16.69W (11.1%). The weight of most relevant embedded computing platforms ranges from 40g (Raspberry Pi 3) to about 180g (Intel NUC Board NUC5I3MYBE).

Let's have a look at the power consumption of embedded computing platforms to determine if this influences the total UAV power consumption more than adding extra weight or not. Typical computing platforms intended for embedded applications tend to not use more than 15W under load (Intel NUC Board NUC5I3MYBE for instance), many even settle with a lot less (RaspBerry Pi 3 under load uses about 6.7W). Taking into account the previous thoughts and the fact that we need as much CUDA power as we can get in an embedded package, the choice for the NVIDIA Jetson TX2 was quickly made. At 85 grams (without expansion board or cooling fan) it presents 256 CUDA cores (1.5TFLOPS) and runs at either 7.5W (peak efficiency) or 15W (peak performance). It is twice as efficient as its predecessor (Jetson TX1) and features specifications that are currently unmatched in developer boards.

4.4. Decision support system

The automated decision support system's role is to decrease the information load decision makers and actors on scene face during emergencies. In stressful and volatile situations details are easily overlooked, or there simply isn't enough time to assess the full context of the incident. The decision support system can provide helpful insights based on information gathered by the UAV or other data sources. An example of how this could look like can be found in Figure 3. The bold red line represents an underground gas pipe, the location of which can be acquired from an online API, or an offline cached database. A slightly transparent red area represents a security perimeter around the gas pipe that should be kept clear during incidents and the green marker represents an emergency responder, located by the detection setup running on our autonomous UAV. The event-based decision support engine reasons that a person is inside a restricted area and generates a warning to notify whoever is coordinating the operation of this incident. Many other incidents can be defined and processed in the same fashion: the UAV could autonomously search for wounded persons in the area, track the location of rescue workers and rescue vehicles on site etc. An evaluation of the decision support system is out of the scope of this paper, in what follows we focus on achieving a real-time object detection on-board the UAV.

5. Evaluation

As the viewpoint of a low-altitude UAV distinctly differs from a traditional ground perspective, it makes sense to evaluate the different YOLO configurations using appropriate image and video material. The UAV123 data set [16] provides exactly that. 123 partially annotated (only one person is annotated, we filter out sequences where multiple people are visible to calculate precision) video sequences in 16:9 format (1280x720) captured from a UAV provide the perfect benchmark material. The following subsections discuss the various parameters we have analyzed.

5.1. Input resolution

The input resolution of the Darknet neural network YOLOv2 uses defaults to 416x416. Every image subject to training or inference is resized to this raster. Increasing these dimen-

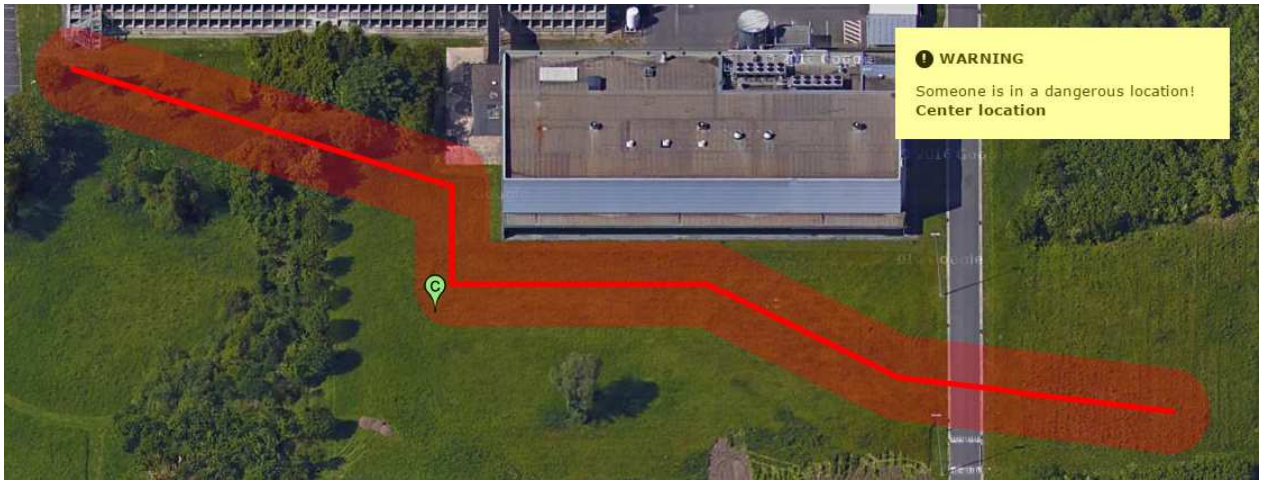

Figure 3. Example decision support interface

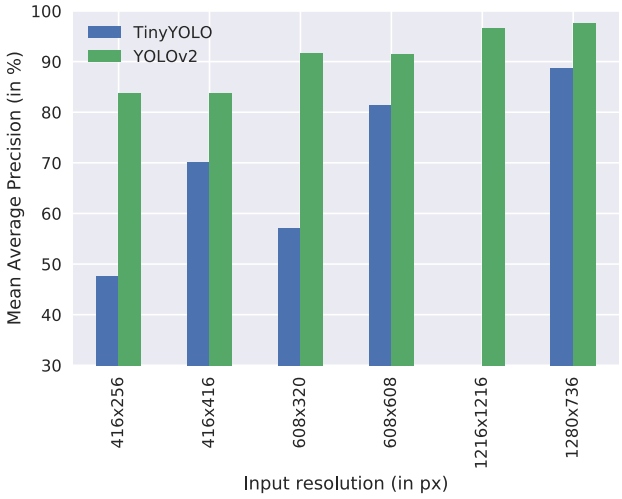

Figure 4. Input resolution compared to accuracy

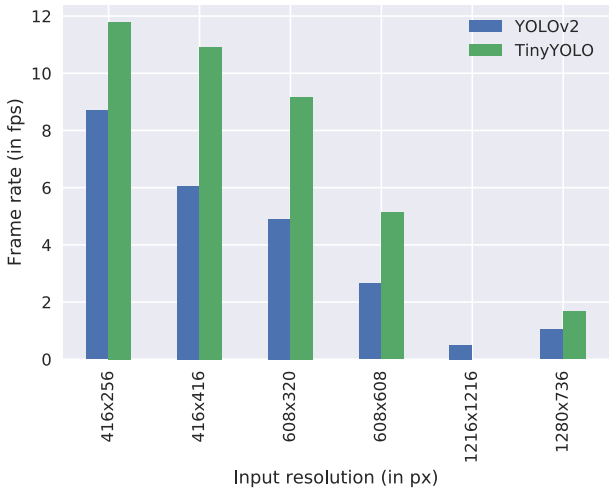

Figure 5. Influence of input resolution on detection frame rate

sions will increase the time required for inference, but also results in a greater detection accuracy. This allows us to define either a low frame rate, high accuracy system, a high frame rate, lower accuracy system or anything in between, depending on the application. As every image or video frame needs to be resized to the defined input resolution, the dimensions of the input frame relative to the raster will have a major impact on the attainable frame rate. Figure 4 illustrates the increasing detection accuracy with increasing input resolution. From the figure, it follows that for YOLOv2 the accuracy goes up only marginally if the width is maintained, but the height increased to form a square input raster. This can be explained by the fact that the source images are all in 16:9 format. Resizing a rectangular image to a square raster introduces irrelevant information (YOLOv2 does not crop while resizing) that doesn't help detection accuracy. This deduction is not valid for TinyYOLO however, as a rectangular input resolution of 416x256 or 608x320 gravely harms the mAP.

Increasing the detection accuracy by means of a larger input raster comes at a hefty price as Figure 5 illustrates. Comparing the lowest accuracy (83.67% @ 416x256) to the best value (97.67% @ 1216x1216), we see the YOLOv2 frame rate drop from 6fps to 0.497fps, a 12-fold decrease. The Jetson TX2's limited memory prevented us from running TinyYOLO at a 1216x1216 resolution, but looking at the other results we see a comparable decrease from 11.80fps (416x256) to 1.67fps (1280x736).

Combining results from both previous plots yields an interesting conclusion: the frame rate difference between an input resolution of 608x320 and 416x416 is smaller than between 608x320 and 608x608, while the detection accuracy of 608x320 over 416x416 is greatly increased and there is almost no difference between 608x320 and 608x608. As such, when configuring a YOLOv2 detector it is advised to set the aspect ratio of the input raster to that of the images or video stream the network will process, in order to obtain an optimal frame rate and detection accuracy. In the case for TinyYOLO, a rectangular input raster should be chosen.

5.2. Video input dimensions

To evaluate the exact effects the input video dimensions have on the frame rate, we chose a NASA 4K video[3] (3480x2160, 24 frames per second, 20.71Mbps bit rate, Public Domain Mark 1.0) named 'Red Lettuce', as it displays multiple categories we can detect using the default COCO and VOC trained models for YOLOv2 and TinyYOLO. We have subsequently downscaled this video to other 16:9 formats using ffmpeg and let both YOLOv2 and TinyYOLO process this batch. The input resolution was configured at 416x416 for every run. Figure 6 illustrates the results of this evaluation.

A first obvious conclusion we can draw, is that the frame rate goes down as the input video resolution goes up. This effect is not the same for YOLOv2 and TinyYOLO however: TinyYOLO is affected by this in a greater way. The TinyYOLO COCO frame rate drops from 14.87fps (256x144) to 4.91fps (3840x2160) a 66.98% decrease, while YOLOv2 COCO goes from 6.97fps (256x144) to 4.25fps, representing a 39.02% decrease. A second observation shows that the inference time greatly depends on the model size. Comparing the VOC (20 categories), COCO (80 categories) and YOLO9000 (9142 categories) models for YOLOv2, we see that as the amount of detectable categories increases, more time is required to perform the detection. As the amount of detectable categories in our use case will be well below 100, this will not pose a major influence.

5.3. ACF compared to Yolo

To see how YOLO compares to traditional detection methods, we compare the different configurations of YOLO and

[3] https://archive.org/details/NASA-Ultra-High-Definition

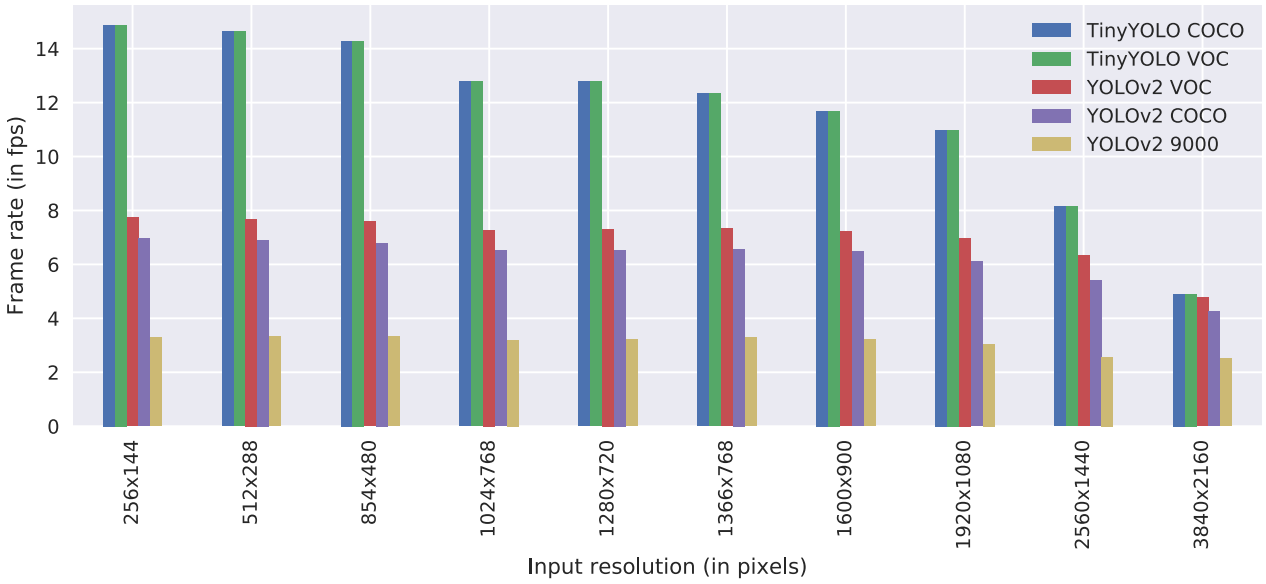

Figure 6. Influence of video dimensions on detection frame rate

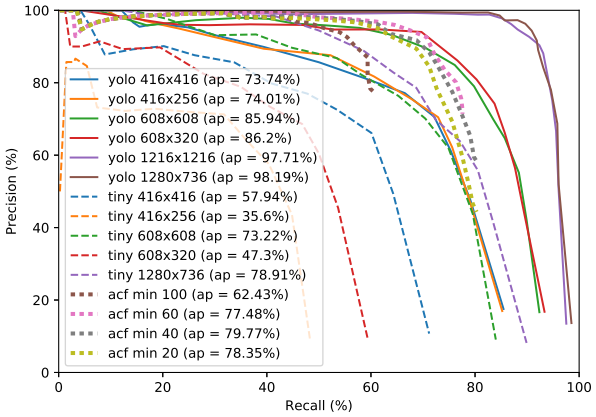

Figure 7. Comparison of the ACF detector to YOLO

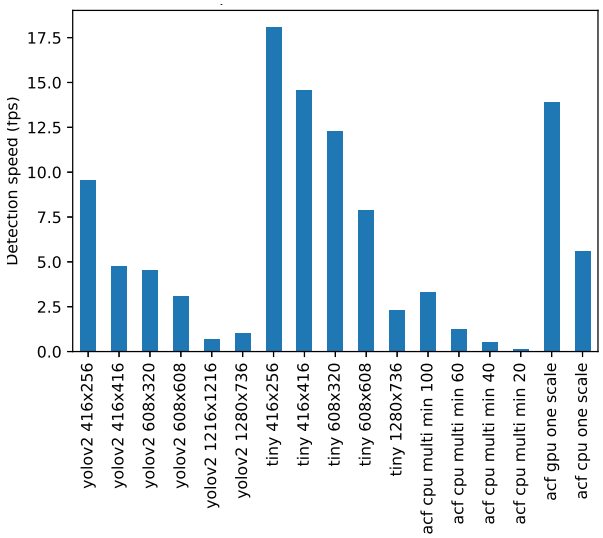

Figure 8. Comparison of different detectors in terms of speed

TinyYOLO in this section to the aggregate channel features (ACF) detector proposed by Dollár et al. [4]. We used our own implementation of ACF which was ported from Matlab to C++ for better performance. We also created an experimental GPU implementation of the ACF detector which can currently only evaluate features at a single layer in the feature pyramid, but can still give us an idea about how well ACF would scale to GPU.

The results in terms of detection accuracy can be seen in Figure 7. ACF (evaluated with a minimum detection scale of 20 pixels) shows similar performance to TinyYOLO evaluated at a resolution of 1280x736 pixels and YOLOv2 using an input resolution that lies between 416x256 and 608x320 pixels.

Figure 8 compares the speed of different detectors. The CPU multi scale ACF detector with minimum detection height of 100px is about as fast as YOLOv2 with a resolution of 608x608. Looking at the accuracy of both detectors, it follows that YOLOv2 outperforms ACF with an average accuracy of 85.9% vs. 62.4% suggesting that at least compared to the CPU version of ACF, YOLOv2 performs better at the same frame rate. Also note that the YOLO 608x320 variant attains about the same accuracy and is faster. Lowering the minimum height to 40px gives us the best accuracy for the ACF detector (ap = 79.7%), the closest corresponding YOLO detectors are YOLOv2 608x320 and YOLOv2 416x256 with an average precision of 86.2% and 74.0% respectively. The ACF detector reaches a mean frame rate of 0.54 fps while the YOLO detectors reach a frame rate of 4.53 and 9.57 fps respectively.

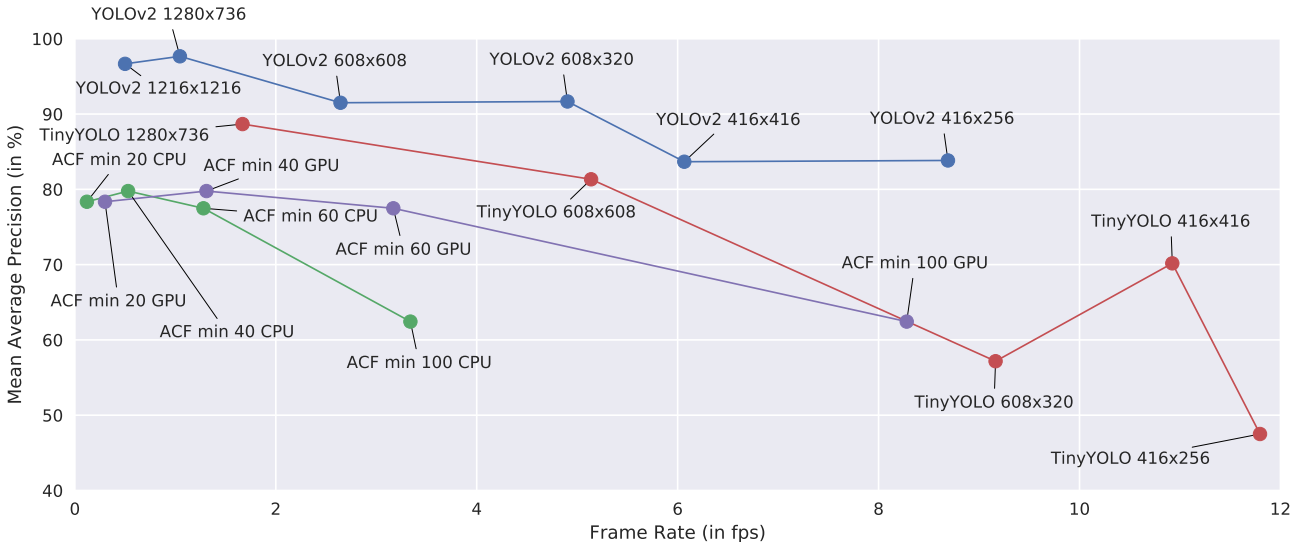

Figure 9. Mean Average Precision vs. frame rate for the discussed object detection algorithms running on an NVIDIA Jetson TX2

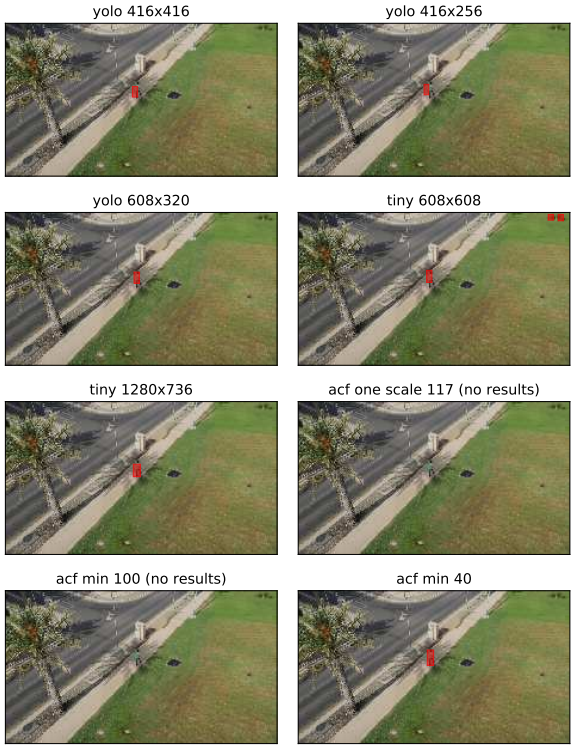

Figure 10. Visual comparison a selection of detector on a single frame of the UAV123 dataset.

Previous results did not run on a GPU, which is a really powerful component on the Jetson TX2 when compared with the ARM CPU. To get equivalent results as YOLOv2, a GPU ACF implementation would need a speedup of at least $10\times$ compared to CPU. Comparing the one scale implementation of ACF suggests that with the current implementation we can get a speedup of $2.5\times$. The biggest bottleneck here is the traversal of decision trees, which is a memory bound problem and thus hard to speed up on GPU. An additional speed increase in ACF could consist of the addition of scene constraints by using the height and orientation of the drone, and assuming a flat surface underneath. We can vastly narrow down the amount of scales that need to be searched and thus speed up the detector even more. Figure 7 shows the accuracy of the ACF detector at one scale, where we set the scale to 117 pixels (the average height of a person in the UAV123 dataset). The poor performance shows that it is indeed necessary to detect on multiple scales when using the ACF detection. Alternatively, with the knowledge of the earlier drone measurements we can greatly increase this accuracy by detecting on the correct scale. Of the currently evaluated detectors, the YOLOv2 detector seems to bring the best accuracy combined with the highest frame rate. Figure 10 shows a visual comparison of a selection of the evaluated detectors.

6. Conclusion

We have presented and evaluated our approach for a real-time object detection system on-board an autonomous UAV. Multiple configurations are available, depending on the frame rate and detection accuracy the final application requires. The presented evaluations can help configure future applications and demonstrate what is currently feasible using off-the-shelf hardware. The modern, neural net based YOLOv2 algorithm attains higher frame rates and detection accuracy results than leading CPU based algorithms. Figure 9 presents a concluding view of how the different algorithms perform under varying conditions on an NVIDIA Jetson TX2, comparable to how Figure 1 did for several algorithms running on an NVIDIA Titan X (GM200).

References

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv:1603.04467*, 2016.

[2] S. Candiago, F. Remondino, M. De Giglio, M. Dubbini, and M. Gattelli. Evaluating multispectral images and vegetation indices for precision farming applications from uav images. *Remote Sensing*, 7(4):4026–4047, 2015.

[3] N. Dalal and B. Triggs. Histograms of oriented gradients for human detection. In *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on*, volume 1, pages 886–893. IEEE, 2005.

[4] P. Dollár, R. Appel, S. Belongie, and P. Perona. Fast feature pyramids for object detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2014.

[5] P. Dollár, Z. Tu, P. Perona, and S. Belongie. Integral channel features. 2009.

[6] P. Felzenszwalb, D. McAllester, and D. Ramanan. A discriminatively trained, multiscale, deformable part model. In *Computer Vision and Pattern Recognition, 2008. CVPR 2008. IEEE Conference on*, pages 1–8. IEEE, 2008.

[7] A. Glaser. Dji is running away with the drone market. https://www.recode.net/2017/4/14/14690576/drone-market-share-growth-charts-dji-forecast, 2017. [Online; accessed 14-04-2017].

[8] I. Hong, M. Kuby, and A. Murray. A deviation flow refueling location model for continuous space: A commercial drone delivery system for urban areas. In *Advances in Geocomputation*, pages 125–132. Springer, 2017.

[9] D. Hulens and T. Goedemé. Autonomous flying cameraman with embedded person detection and tracking while applying cinematographic rules. In *Proceedings CRV 2017*, 2017.

[10] D. Hulens, J. Verbeke, and T. Goedemé. Choosing the best embedded processing platform for on-board uav image processing. In *VISIGRAPP*, pages 455–472. Springer, 2015.

[11] K.-H. Kim, S. Hong, B. Roh, Y. Cheon, and M. Park. Pvanet: Deep but lightweight neural networks for real-time object detection. *arXiv preprint arXiv:1608.08021*, 2016.

[12] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[13] S. Lange, N. Sunderhauf, and P. Protzel. A vision based onboard approach for landing and position control of an autonomous multirotor uav in gps-denied environments. In *Advanced Robotics, 2009. ICAR 2009. International Conference on*, pages 1–6. IEEE, 2009.

[14] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg. Ssd: Single shot multibox detector. *arXiv preprint arXiv:1512.02325*, 2015.

[15] L. F. Luque-Vega, B. Castillo-Toledo, A. Loukianov, and L. E. Gonzalez-Jimenez. Power line inspection via an unmanned aerial system based on the quadrotor helicopter. In *MELECON, 2014 17th IEEE*, pages 393–397. IEEE, 2014.

[16] M. Mueller, N. Smith, and B. Ghanem. A benchmark and simulator for uav tracking. In *European Conference on Computer Vision*, pages 445–461. Springer, 2016.

[17] A. Obukhov. Haar classifiers for object detection with cuda. *GPU Computing Gems Emerald Edition*, pages 517–544, 2011.

[18] J. Redmon. Darknet: Open source neural networks in c. *http://pjreddie. com/darknet*, 2016, 2013.

[19] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 779–788, 2016.

[20] J. Redmon and A. Farhadi. Yolo9000: better, faster, stronger. *arXiv preprint arXiv:1612.08242*, 2016.

[21] J. Ren, X. Chen, J. Liu, W. Sun, J. Pang, Q. Yan, Y.-W. Tai, and L. Xu. Accurate single stage detector using recurrent rolling convolution. *arXiv:1704.05776*, 2017.

[22] S. Ren, K. He, R. Girshick, and J. Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. In *Advances in neural information processing systems*, 2015.

[23] G. Sona, D. Passoni, L. Pinto, D. Pagliari, D. Masseroni, B. Ortuani, and A. Facchi. Uav multispectral survey to map soil and crop for precision farming applications. *Internation Archives Of The Photogrammetry, Remote Sensing And Spatial Information Sciences*, 41:1023–1029, 2016.

[24] C. Szegedy, S. Reed, D. Erhan, D. Anguelov, and S. Ioffe. Scalable, high-quality object detection. *arXiv preprint arXiv:1412.1441*, 2014.

[25] N. Tijtgat, B. Volckaert, and F. De Turck. Real-time hazard symbol detection and localization using uav imagery. In *2017 IEEE 86th Vehicular Technology Conference (VTC-Fall)*, pages 1469–1474. IEEE, 2017.

[26] S. Tripathi, G. Dane, B. Kang, V. Bhaskaran, and T. Nguyen. Lcdet: Low-complexity fully-convolutional neural networks for object detection in embedded systems. *arXiv preprint arXiv:1705.05922*, 2017.

[27] P. Viola, M. J. Jones, and D. Snow. Detecting pedestrians using patterns of motion and appearance. In *null*, page 734. IEEE, 2003.

[28] B. Wu, F. Iandola, P. H. Jin, and K. Keutzer. Squeezedet: Unified, small, low power fully convolutional neural networks for real-time object detection for autonomous driving. *arXiv preprint arXiv:1612.01051*, 2016.