6.869 Final Project, Fall 2017
Object Detection on an Embedded GPU for Autonomous Driving

Nathan Monroe, 6.869
Kerberos: monroe
monroe@mit.edu

Collaborated with: Mark Mazumder, 6.869
Kerberos: ma23705@mit.edu
mazumder@ll.mit.edu

Abstract

We explore the application of SSD-MobileNet to an embedded system in the context of small mobile autonomous vehicles. We demonstrate real-time object detection on-device in a mobile robotic platform, and the ability to infer object locations and categories at approximately seven frames per second. We incorporate object detection into a motion control loop on a small autonomous vehicle, for recognizing and tracking specific objects. We improve stability and robustness by applying filtering techniques, control loop tuning, and a custom training dataset for object recognition to optimize for pose, scale, and illumination invariance. The vehicle is capable of tracking objects in the presence of significant visual clutter and a variety of lighting conditions. We further apply the object detection model in the context of gesture recognition, allowing a user to command the vehicle's velocity by simple hand motions. We demonstrate the generality of object detection algorithms in embedded computing by applying an on-device model for license plate detection and recognition. Demo videos are available at `https://photos.app.goo.gl/TG3DIh07jPN0ZY3F2` *and in the included supplemental material [1].*

1. Introduction

Small mobile robotic platforms are constrained by their need to minimize size, weight, and power consumption. This presents challenges to autonomous computer vision tasks such as object recognition and tracking. Deep learning has enabled many recent advances in computer vision, but these algorithms require significant computational resources. Deep neural network (DNN) models often consist of tens of millions of parameters, loaded as weights into a computational graph. To efficiently make predictions, these graphs must reside entirely in RAM and inputs must be processed via highly parallelized load-store and arithmetic operations. GPUs are ideal devices for such computational needs, but most GPUs used in deep learning are desktop or server-class, with corresponding size and power requirements. Small mobile robots are ill-equipped to incorporate these devices on-board. Distributed computing is one approach to incorporating DNN models as part of a real-time control objective on a mobile robot, for instance by streaming a robot's camera feed over a network to a dedicated GPU. However, this requires a robust low-latency network, often limiting applications to indoor research laboratories with additional mitigations to reduce wireless interference, since a brief interruption in a camera feed may cause controller instability. Recent research efforts such as Google's 2017 MobileNets [8] seek to reduce the computational requirements of deep learning algorithms in order to target mobile GPUs, which are present in many smartphones and other portable electronic devices such as video game systems, offering only a fraction of the RAM and compute cores present on desktop/server GPUs. These embedded GPUs can be integrated into mobile robotic platforms. Our work aims to apply these recent advances in resource-efficient deep learning to a small mobile robot for real-time object recognition and tracking.

We incorporated object detection into an autonomous control loop on a small robotic car, which we trained to recognize and follow a specific person at a given distance as they walk around. The car platform we used in our experiments provides a 720p camera, an embedded GPU, and a simple motor API which accepts velocity and steering angle commands (Sec. 3.1). Due to the car's small size, the camera is low to the ground. Therefore, we trained a TensorFlow-based object recognition model to detect and discriminate between the shoes worn by the two authors (Sec. 3.2). This shoe detection model outputs bounding boxes, which are used as inputs to the robot's velocity and speed controllers. (Sec. 3.3).

To enable real-time object detection on our embedded GPU, we trained SSD-MobileNet using the TensorFlow Object Detection API (Sec. 1.1). This highly optimized network requires only 22MB of weights and can infer objects while using under 2GB of GPU memory allocations. This enabled us to achieve real-time inference for integration into our motor controllers. Our recognition and tracking controller is able to continuously follow a specific person in a variety of lighting and environmental conditions. We demonstrate it tracking a person as they walk through various real-world environments, in the presence of significant clutter visible on the ground (such as table legs, boxes, and research laboratory equipment).

We also demonstrate the incorporation of this object detection network into a gesture recognition controller. In our implementation, the car accepts commands to drive forwards or backwards in response to hand motion when both hands are present in frame (Sec. 3.4). Finally, we demonstrate that modern embedded CPUs are also appropriate targets for many object recognition applications which are not based on DNNs . We demonstrate this by applying an existing model for license plate detection and character recognition on the platform's embedded CPU (Sec. 4.4).

1.1. Object Detection on Embedded GPUs

We performed object detection using TensorFlow's Object Detection API [14] and a variant of the Single-Shot Multibox Detector (SSD) network [11]. SSD is an object detection network which appends convolutional feature maps after the initial layers of an image classification network. These appended convolutional layers have outputs of different sizes; activations correspond to detections of objects at different scales. For each feature map scale, SSD predicts an object's presence among several default boxes tiled spatially across the image. Each default box contains a prediction for each class at multiple aspect ratios. These play a similar role to the anchor boxes in the Faster-RCNN object detection model [11]. Non-maximum suppression is used to limit the number of bounding boxes predicted at a given spatial location to a single box with the highest confidence.

The original SSD implementation used the initial layers of the VGG image classification network [11]. The large network size limits SSD's ability to perform real-time detection to desktop and server-class GPUs such as the Nvidia TITAN X. In 2017, Google released the MobileNet architecture [8], which is intended for mobile GPU applications. MobileNet decomposes a standard convolution operation into two convolutions: a channel-based depthwise separable filter and a single-point 1x1 filter that combines channels.

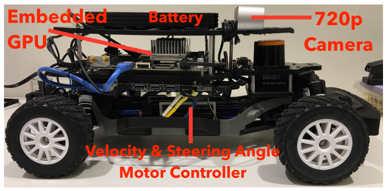


Figure 1: MIT RACECAR programmable vehicle.

The separable filter simplifies a 2D MxN convolution by approximating it as two 1D (Mx1 and Nx1) convolutions, reducing the number of multiplications and additions necessary. This comes at the expense of fewer tunable kernel parameters. Unlike a traditional convolutional kernel, separable kernels are applied independently to each of the three color channels. MobileNet exploits matrix sparsity and can be further optimized using optional width and resolution hyperparameters, which reduce the number of channels or the dimensions of the input image.

Our implementation uses SSD-MobileNet from TensorFlow's Object Detection API [14]. As described in the MobileNet implementation [8, 9] (Sec. 2), after replacing the VGG layers in SSD with MobileNet's depthwise-convolutional layers, SSD-MobileNet achieved an mAP of 19.3%. This is close to SSD-VGG's mAP of 21.1% (based on the COCO primary challenge metric of AP at IoU of [0.50:0.05:0.95] [3]), but SSD-MobileNet only requires 3.4% of the computational resources of SSD-VGG and 20% of the parameters (6.8M compared to 33.1M). As a result, SSD-MobileNet was capable of performing realtime inference on the Nvidia Tegra X1 embedded GPU used in our implementation (Sec. 3).

1.2. Division of Work

Both teammates contributed equally to the project. In particular, equal effort was applied in high level project implementation phases, image collection and labeling, robotics debugging, object detection deployment, high level integration, and generation of presentation slides and this paper. I (Nathan) was the leader on robotics control tuning, license plate recognition and memory portions. Collaborator (Mark) was the leader on object detection network training and the gesture recognition portions.

2. Related Work

The domain of visual servoing involves the use of camera inputs to control motor outputs, often to track an object in frame. A common approach for visual servoing is color thresholding [10]. In this approach, images are segmented based on colors, often following transformation into color spaces such as HSV. This approach is computationally efficient for embedded devices. However, HSV thresholding approaches are often unable to generalize to arbitrary real-world settings, where varying illumination and the potential presence of similar-colored objects adversely impact the detection rate.

Deep learning approaches to object detection have yielded high accuracy in generalizable settings, but only a subset of recent networks such as R-FCN, SSD, and YOLO architectures have demonstrated sub-second frame-rates [9]. These networks still target desktop or server-class GPUs to achieve high frame rates for video processing. The

YOLOv2 network has been adapted to run on the Nvidia Tegra X1 embedded GPU by [12], after reducing the number of parameters used in the original YOLOv2 architecture. However, in order to achieve a real-time framerate on the embedded GPU, motion estimation was applied as a filter and only a subset of remaining frames were used for YOLO inference. In contrast, our system applies inference at a steady rate of seven FPS.

The comparison study of several object detection architectures in [9] proposes a unifying implementation of Faster R-CNN, R-FCN, and SSD as three meta-architectures for object detection, with several feature extraction networks available for incorporation within each object detector meta-architecture, such as VGG-16, MobileNet, Inception variants (e.g., V2), and ResNet variants. The combination of a low number of parameters and high accuracy exhibited by SSD-MobileNet is well-suited for embedded and mobile GPU applications. The TensorFlow Object Detection API [14] implements the meta-architectures proposed in [9], and we have relied on TensorFlow's implementation for our work.

3. Approach

This section details the vehicle and embedded hardware platform underlying our implementation. We describe our training methodology for our three object classes (two types of shoes, plus hands) using SSD-MobileNet, and the control algorithm we use to command the car to autonomously track specific people. Finally, we present the mechanism we developed for simple gesture-based control.

3.1. Hardware Platform

The hardware platform used in our implementation is the MIT RACECAR [2], a small programmable vehicle with a wheelbase of 55cm x 29cm (Fig. 1). The vehicle was developed for classroom instruction in Robotics: Science and Systems (6.141/16.405).[1] Our onboard computation was facilitated by the Nvidia Jetson TX1 [2] present on the vehicle. This is a battery-powered embedded computer which offers an ARM Cortex-A57 CPU and a mobile GPU, identical to the one used in the popular Nintendo Switch portable video game system. The TX1 has 4GB of RAM which is shared by both GPU and the CPU, and runs Ubuntu 16.04 and ROS Kinetic [2]. The motors are controlled with the VESC electronic speed controller API, which accepts velocity and steering angle commands over USB and exposes them through a ROS python interface. The car offers two cameras, but our implementation only used the left camera. This camera captures RGB images at 30 frames per second at a resolution of 1280x720. This camera was also used to collect training images for subsequent manual labeling.

[1]Neither author has taken this class, and we are grateful for the opportunity to borrow the platform.

3.2. Object Detection Training

To achieve real-time object detection on an embedded platform in the context of autonomous control, we selected SSD-MobileNet due to its high accuracy and minimal model size (Sec. 1.1). Our system design for recognizing and autonomously following specific objects consists of two main components. SSD-MobileNet detects bounding boxes for a specific object in the camera frame. This feeds into a robotic control loop which implements our tracking algorithm (Sec. 3.3).

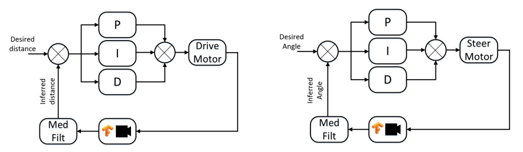
We fine-tuned SSD-MobileNet using pretrained weights on the COCO dataset [3], which were downloaded from Google [14]. The COCO model does not contain a generic shoe category. We therefore created our own two-class training corpus of 1070 images of the authors' shoes taken as each author walked in view of the car's camera. These images were manually labeled using the VIA tool [5]. 161 images were used for validation. Over 90% of the images contain two bounding boxes representing a pair of shoes, unless one foot occluded the other in the middle of a step. To encode pose and scale invariance in our training set for each shoe object category, we walked in multiple random directions and at multiple distances (between 0 and 7m) in front of the camera.

We provide the key parameters used for fine-tuning as follows (all parameters are reported in [1]). Images were rescaled from 1280x720 to an input size of 300x300. Six layers were used for SSD's convolutional feature maps (Sec. 1.1), evenly scaled between 0.2 and 0.95 and mapped to SSD's default boxes, and five aspect ratios between $\{0.333, 0.5, 1.0, 2.0, 3.0\}$ were predicted for each default box. Non-max suppression was configured to use an IoU threshold of 0.6. MobileNet was configured to use the default maximum depth multiplier of 1.0. The learning rate was fixed at 0.004, and data augmentation consisted of random horizontal flips and crops. The model was configured to return 100 bounding boxes per inference with corresponding confidence scores. Section 4.1 provides the thresholds we used to filter these confidence scores.

The 4GB of RAM on our embedded platform is shared between the CPU and GPU. The operating system, robot control libraries, and camera allocate approximately 2GB of RAM. Critically, our embedded kernel configuration does not support swap space. By default, TensorFlow attempts to allocate all available GPU RAM, which immediately exhausts user memory and the process is resultantly killed by the Linux Out-Of-Memory (OOM) Killer. Therefore, we empirically limited TensorFlow's memory allocator to 1.3GB. We found this sufficient for the SSD-MobileNet computational graph to reside in RAM. We also had to recompile the embedded Linux kernel in order to enable concurrent support for Nvidia CUDA 8.0 (a TensorFlow library dependency) and the ACM kernel module which enables the velocity and steering controller USB device.

3.3. Autonomous Tracking

Autonomous tracking is derived from object size and location output from the object detector. This is realized as two PID control loops, one each for steering and velocity (Fig. 2). The steering control loop uses as an error function the mean center location of detected objects of the desired class. It attempts to center this mean location in the camera's field of view. Similarly, the velocity PID loop controls the car's drive motors. This controller attempts to maintain a constant distance from the desired object, using object height as a proxy for distance. The controller's error function in this case is the difference between mean height of detected objects and desired height. The resulting behavior of the two controllers is a vehicle which follows a specified object at a fixed distance. Both controllers incorporate temporal and spatial median filtering on object detections to mitigate detection noise and spurious detections which are inherent in an inference-based control loop.



(a) Velocity Controller (b) Steering Controller

Figure 2: PID control loops for both drive velocity and steering, allowing the vehicle to follow a specified object at approximately a specified distance.

3.4. Integration into Gesture Recognition

We incorporated SSD-MobileNet into a second application running on-board the car: gesture recognition for commanding velocity. The gesture we selected was the relative position between two hands. The vehicle is commanded to drive forwards if the user raises their left hand up in relation to their right hand, with both hands present in the camera frame. Conversely, the car drives backward if the right hand is raised above the left. We trained a one-class model on 273 images of the authors' hands with open palms, captured with the car's camera and labeled using VIA [5]. Forty nine images were used for validation, with 90% of the images containing two hands. The model uses the same configuration parameters as the shoes model in Sec. 3.2, with the exception of only predicting a single class.

To command velocities when two hands are detected, the pose of the hands is estimated using the relative angle between them. First, the midpoint of each bounding box (corresponding to each detection) is calculated, and a line segment is calculated between these two midpoints. The angle between the hands is estimated using the arccosine of the dot product between a unit vector parallel to the x-axis of the camera and a vector from the midpoint of the line segment to the midpoint of the rightmost bounding box (with respect to the camera frame). Fig. 6 in Sec. 4.3 provides a visual interpretation of this pose estimation approach. If more than two bounding boxes are detected for a given confidence threshold, our implementation performs spatial filtering by selecting the two bounding boxes which are physically the lowest in the frame.

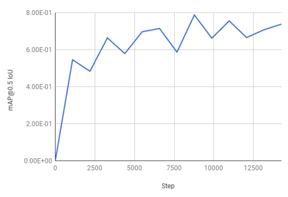
4. Experiments

4.1. Object Detection Results

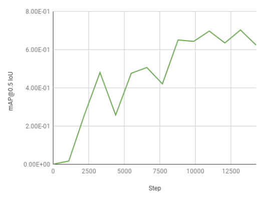
Our two-class SSD-MobileNet object detection model was fine-tuned for 14268 steps after initializing the model with weights trained on the COCO dataset [14], as described in Sec. 3.2. To evaluate our model's performance in detecting bounding box locations, we rely on the industry standard mean Average Precision (mAP) metric [6] at 0.5 Intersection-over-Union (IoU). IoU refers to the number of pixels in the intersection between a predicted bounding box and a ground-truth bounding box from our manually labeled dataset. With false positives defined as predicted bounding boxes that do not correspond to a manually labeled bounding box, and false negatives defined as a ground-truth box which does not overlap with any predictions, a precision-recall curve results. Average Precision approximates the area under the precision-recall curve, and the mean is reported across all classes and validation examples for an IoU of 0.5 in Fig. 3. Figs. 3a and 3b report the mAP across all validation examples for each author's own shoe class. Thresholds for successful detection were determined empirically to be 0.15 and 0.2 for the two shoe classes chosen. Low thresholds were selected to maximize detection distance, and increase robustness in the presence of varied lighting conditions. In addition, the detector only returns the two most confident detections, so the risk of spurious detection is low. The object detector was found to reliably detect shoes with a height of approximately 25 pixels, or approximately 5 meters from the car.

4.2. Autonomous Control Results

Control loops for autonomous tracking were tuned according to standard control design practices for maximum convergence time while minimizing overshoot and oscillation in the context of loop stability. Proportional, Integral and Derivative gain values for the two controllers are as follows: for steering, $P = 0.0005, I = 0, D = 0.0001$, and for velocity, $P = 25, I = 0, D = 0$. In addition, a median filtering block was included in the object detector's output for robustness and stability. The length of the median filter represents a tradeoff between noise immunity and response time. The optimal filter length was empir-



(a) Mark Shoes mAP



(b) Nathan Shoes mAP

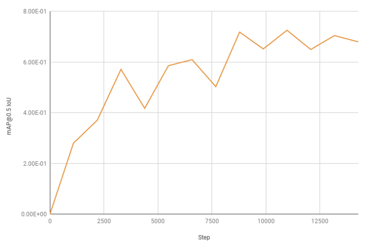
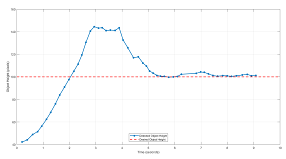
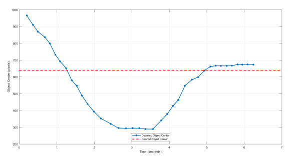


Figure 3: mAP@0.5IoU for both shoes after 14K steps.

ically set at 3. At an inference rate of approximately 7 FPS (143ms per frame), this represents a filtering delay of approximately 428ms. Representative step response plots of the two controllers to inputs in distance and angle are given in Fig 4. Parameters were tuned for acceptable convergence times and minimal overshoot. Drive velocity was limited to a maximum magnitude of 0.5m/s. Convergence time can be increased at the expense of this safety margin, and we note this limit imposes nonlinear velocity control. A nonlinear relationship additionally exists between bounding box height and distance from the car, due to the trigonometry of the object's apparent height at different distances. An optimal height for shoes was found to be 100 pixels, or an approximate distance of 30cm based on known shoe heights. Overshoot in distance is equal to approximately 45px, or approximately 8cm. Due to the car's inability to turn while stationary, the steering controller occasionally exhibits a slight convergence offset if the desired height is found before the object is fully centered (Fig. 4b), a target for future optimization.



(a) Velocity Step Response



(b) Steering Step Response

Figure 4: Step responses of two example runs with an objective (y-axis) in pixels, for velocity and steering PIDs.

4.3. Gesture Recognition Results

Hand detection for gesture recognition-based control exhibited high accuracy and robustness due to it's having a single class output, as well as the relatively unique shape of a human hand. The detector is robust to varied lighting conditions and hand poses. Across 49 validation images (most containing two hands), a mAP@IoU=0.5 of 0.9085 was achieved after fine-tuning on pretrained COCO weights for 10262 additional steps. During inference, the confidence threshold for hand detection was empirically set at 0.7. The device was able to achieve the same inference rate as the shoe detector (seven frames per second), with framerate being dominated by SSD-MobileNet's inference time. A delay of approximately 428 milliseconds exists between gesture input and appropriate motion output due to the median filter implementation, as described in Sec. 3.4. While this implementation only allows for gestural control of linear velocity, it could easily be extended to more advanced control such as steering. In this implementation, consistent detections can be made with a minimum bounding box size of 34 pixels, corresponding to a maximum distance of approximately four meters.

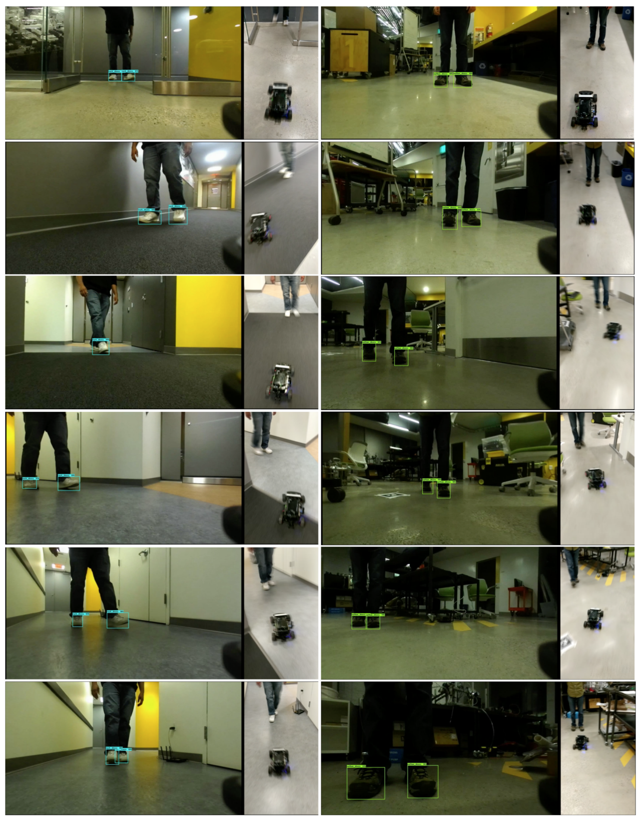


Figure 5: Two walking sequences with autonomous tracking. Left: detections, right: 3rd-person view of the car.

4.4. License Plate Recognition

In order to demonstrate the generality of object detection algorithms in embedded computing, we demonstrate an entirely different detection algorithm in a different context: license plate recognition. We applied license plate recog-

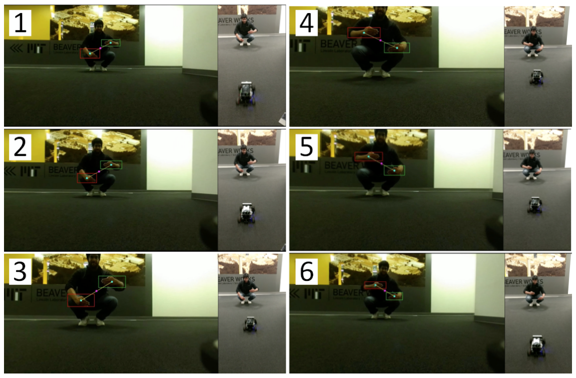


Figure 6: Gesture recognition for velocity control.

nition techniques on the MIT RACECAR using the openALPR library [4]. The openALPR library uses a pipelined approach for detecting and localizing license plate edges, finding strings of character blobs, homography transformations for perspective warping, optical character recognition on an individual character basis, and regular expression matching to known license plate patterns. OpenALPR leverages the Tesseract OCR library [13] for character feature extraction. In contrast to Mobilenet's convolutional neural network architecture, OpenALPR uses Cascade Support Vector Machines [7] for object recognition. Applying this approach, we were able to achieve real-time on-device license plate localization and detection with multiple simultaneous license plates. Fig. 7 provides an example inferred result. Potential applications include security monitoring and autonomous driving.

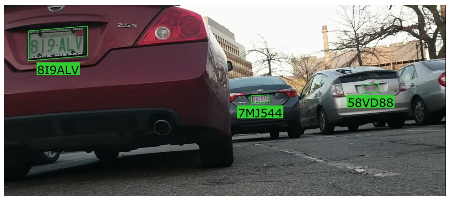


Figure 7: License plate inference examples

5. Conclusion

We have demonstrated the application of real-time object detection on an embedded device in a robotics platform, and the ability to infer object locations and categories at approximately seven frames per second. We incorporate object detection in a feedback control loop for velocity and steering angle to command an autonomous vehicle to recognize and track specific objects, in a processing and memory-constrained embedded computing environment. We optimize for stability and robustness in our custom training dataset, filtering techniques, and control loop tuning. The vehicle is capable of tracking objects in the presence of significant visual clutter and varying illumination. We further applied object detection to gesture recognition, interpreting simple hand motions as velocity commands. Finally, we demonstrated the applicability of non-DNN-based object detection algorithms in embedded computing by using a cascade SVM based model for license plate recognition. These applications demonstrate the potential of object detection networks such as SSD-MobileNet to general tasks in mobile robotics. We propose future work to explore other scaling for SSD-MobileNet beyond 300x300 to optimize for specific camera parameters, desired detection distance, and target framerate. Obstacle avoidance and path planning algorithms could increase tracking performance. Finally, an incorporation of other sensing mechanisms such as LIDAR, depth sensing, and accelerometer measurements into inference and control processes would further extend applicability in real-world settings.

References

[1] 6.869 final project supplementary code: `monroe.zip`.

[2] Mit racecar. `https://mit-racecar.github.io/`.

[3] Coco dataset. `http://cocodataset.org/`, 2017.

[4] Openalpr library. `https://github.com/openalpr/openalpr`, 2017.

[5] A. Dutta, A. Gupta, and A. Zissermann. VGG image annotator (VIA). http://www.robots.ox.ac.uk/ vgg/software/via/.

[6] M. Everingham, L. Van Gool, C. K. Williams, J. Winn, and A. Zisserman. The pascal visual object classes (voc) challenge. *International journal of computer vision*, 2010.

[7] H. P. Graf, E. Cosatto, L. Bottou, I. Dourdanovic, and V. Vapnik. Parallel support vector machines: The cascade svm. In *Advances in neural information processing systems*, 2005.

[8] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *CoRR*, abs/1704.04861, 2017.

[9] J. Huang, V. Rathod, C. Sun, M. Zhu, A. Korattikara, A. Fathi, I. Fischer, Z. Wojna, Y. Song, S. Guadarrama, et al. Speed and accuracy tradeoffs for modern convolutional object detectors. *arXiv preprint arXiv:1611.10012*, 2016.

[10] H. Kadouf and Y. Mustafah. Colour-based object detection and tracking for autonomous quadrotor uav. In *IOP Conference: Materials Science and Engineering*, 2013.

[11] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg. Ssd: Single shot multibox detector. In *European conference on computer vision*, 2016.

[12] M. Shafiee, B. Chywl, F. Li, and A. Wong. A fast yolo system for real-time embedded object detection in video. *arXiv preprint arXiv:1709.05943*, 2017.

[13] R. Smith. An overview of the tesseract ocr engine. In *Document Analysis and Recognition, 2007. ICDAR 2007. Ninth International Conference on*, volume 2. IEEE, 2007.

[14] TensorFlow.org. Tensorflow object detection api. `https://github.com/tensorflow/models/tree/master/research/object_detection/`, 2017.