Object Detection and Imitation Learning in Duckietown

Min Young Chang, Suzie Petryk, Joe Nechleba

Abstract

The crucial first step of any autonomous driving system is to understand the surrounding environment. In Duckietown, the duckies and other duckiebots present hazardous obstacles that the vehicle must not drive into. Thus, object detection is necessary to prevent collisions. In this work, we train a YOLO (You Only Look Once) convolutional neural network for object detection to recognize duckies and duckiebots. The network achieved a 70% average IOU during validation and demonstrates real-time detection of close objects on unseen data. An extension from prior work on imitation learning is also presented. The CNN models were unable to generalize well to unseen Duckietown environments, demonstrating the complexity of the imitation learning task. However, training accuracy improved significantly over time, suggesting its ability to control a duckiebot in familiar environments, such as the same lab where training data was collected.

1. Introduction

Object detection is the task of finding bounding boxes around predefined objects of interest in an image. There must be exactly one bounding box around each instance of an object of interest that appears in the image, and no bounding boxes anywhere else.

Real-world autonomous driving requires the detection of a large number of possible objects, including traffic signs, traffic lights, pedestrians, and other vehicles. Many of these have a large variability in appearance, especially as weather conditions or the road location itself change. An object detection algorithm on real-world vehicles must be robust to these changes in appearance.

In contrast, the Duckietown environment is much less variable in appearance. While the roadmap, lighting conditions, and sideline decorations may change, the roads, traffic signs, duckies, and duckiebots themselves have a fixed design (Liam Paull & others., 2017). This makes the object detection task significantly easier. It allows for a much smaller training set than one needed for real-world driving. This made it feasible to collect a dataset of Duckietown images that was small enough to hand-label duckies and duckiebots, yet large enough to detect most close objects in real time.

The work done in this paper presents a first step[1] in an autonomous driving pipeline for Duckietown created by the Cornell Autonomous Systems Lab.

The layout of this paper is as follows: Section 2 presents the methodology for the object detection work. Results of the training and testing processes are presented in Section 3, followed by a discussion of performance in Section 4. Because the main project focus throughout this semester was object detection, it constitutes the bulk of this paper, followed by Section 6 that describes the extensions on an imitation learning project started in Fall 2018.

The GitHub repository for the object detection work described in this report is found here: `https://github.coecis.cornell.edu/jdn64/duckietown_object_detection`, and the imitation learning GitHub repository containing new extensions is here: `https://github.com/spetryk/imitation-learning-AI-Driving-Olympics`.

1.1. Related Work

Before the use of convolutional neural networks for object detection, the best approaches that relied on classical computer vision techniques (such as SIFT (Lowe, 2004) and HOG (Dalal & Triggs, 2005)) seemed to have plateaued in accuracy on the canonical object detection benchmark of PASCAL VOC. A 30% jump in mean average precison (mAP, a commen metric for object detection performance) was gained in the first application of CNNs to this task (Girshick et al., 2014). This was the "Regions with CNN features" (RCNN) method. First, various region proposal algorithms are used to extract about 2k possible object regions. A CNN is used to extract a feature vector for each of these regions, which is then classified with a support vector

[1] The first step constitutes perceiving the environment, which in this pipeline includes lane detection (described in a different work) as well as object detection.

Figure 1. A sample image from inference on a validation set.

machine (SVM) to give class scores and confidences.

The large per-image computational cost of R-CNN inspired the creation of Fast R-CNN, in which the CNN was run on the entire image once, and then proposals were cropped directly from the resulting feature map (Girshick, 2015). This process was sped up even more with Faster R-CNN (Ren et al., 2015), where a small CNN was used for region proposals. However, the R-CNN family still relies on the pipeline of region proposals to feature extraction to classification, which may lead to inference times that are too long for applications with require fast detection, such as autonomous driving.

The You Only Look Once (YOLO) method streamlines this pipeline into a single CNN (Redmon et al., 2016). First, the image is split up into grid cells. Next, each grid cell predicts a specified number of bounding boxes, along with class scores and confidence for each box. Non-max suppression is then used on the boxes with the highest confidence scores to output one bounding box per object. The network itself has 24 convolutional layers followed by 2 fully connected layers, making its computation time smal enough to be used for real-time detection.

1.2. Problem Statement

The object detection task in the context of Duckietown is defined as follows: given an input image from the front-facing duckiebot camera, output the image coordinates of bounding boxes that tightly enclose each duckie or duckiebot present in the image, and do not appear in the image where there is no duckie or duckiebot (no false positives). There must be exactly one bounding box for each duckie or duckiebot present. Figure 1 presents a sample image from a validation set showing the bounding box detections along with confidence scores for each box.

ROS (Quigley et al., 2009) was used to interface software with hardware on the duckiebot. A Duckietown environment built by the Cornell Autonomous Systems Lab was used for gathering data and testing performance.

2. Methodology

2.1. Data Collection

In order to train the network, a training set consisting of images from the duckietown environment labeled with bounding boxes and classes of objects in each image. Because there was no pre-existing public dataset of this nature, this was gathered by hand. The setup for data collection was as follows: one team member controlled a duckiebot via keyboard, and ran a script that captured rectified images.[2] The other team member moved duckies and a second duckiebot around the environment to gather objects from a variety of positions and orientations. 1012 images with duckies were collected and labeled in this manner, followed by 846 images with both duckies and duckiebots. A labeling tool for bounding boxes was used to vastly speed up the labeling process (Tzutalin, 2015). A data augmentation tool[3] was used to create more training data by varying the brightness, contrast, and color saturation, bringing the total amount of dataset frames to 4894.

2.2. YOLO Training

A Github repository containing the YOLO architecture and training pipeline written in C was used in this work.[4] The dataset was split into 3925 training images and 969 validation images. The network was run with default training parameters[5] and left to train overnight. Analysis of the training and choice of model is found in Section 4.

2.3. Interfacing with Duckiebot

Figure 2 gives an overview of the flow of code between the duckiebot providing the images, ROS master providing data communication, and the GPU/CPU server running the detection code. After starting the necessary programs on the duckiebot, the duckiebot would publish rectified images to a ROS topic. The April tags demo was only needed for the image rectification. We attempted to calibrate the duckiebot (Duckiebot 2 in the lab) to output rectified images, although the homography calibration script threw an error. We contacted the Duckietown coordinators through Slack, who

[2]"Rectified" here means transforming the raw "fisheye" image from the duckiebot into a natural, "flattened" version. This was performed with code from the Duckietown Repository.

[3]https://github.com/mdbloice/Augmentor

[4]https://github.com/AlexeyAB/darknet

[5]https://github.coecis.cornell.edu/jdn64/duckietown_object_detection/blob/master/networks/yolo_duckie_duckiebot_detector/yolov2-tiny-duckie.cfg

notified us that this was a known error with no known solution yet. Thus, we decided to use the April tags demo as a way to activate the publishing of rectified images. A python script was run on an in-lab GPU computer connected to the same network as the duckiebot. It received the rectified images fom the ROS topic and passed them through the trained YOLO model. Next, the image with drawn-on bounding box predictions along with confidence scores was published to another ROS topic, allowing us to see the predictions appear on the duckiebot's feed (through `rqt_image_view`) in real time. Furthermore, each bounding box published to a separate ROS topic giving the its image coordinates, in the form `[left_x top_y width height]`. These bounding box coordinates were used by a next step in the autonomous driving pipeline which predicted the real-world location of the detected objects relative to the duckiebot.

3. Results

3.1. Training and Validation Metrics

The Intersection Over Union (IOU) is a common metric to measure object detection performance. It measures the fraction overlap of the ground truth bounding box with the predicted bounding box for a given object. It ranges from 0 to 1, where 1 is a perfect overlap. Figure 3 gives the average IOU per batch during training. It also gives several points evaluated on validation data. The YOLO training framework did not provide functionality to track validation performance at each batch, and thus the validation points are much more sparse.

In an attempt to avoid overfitting, we chose the model trained for 20,000 batches to deploy to the duckiebot, rather than the final model at 40,000 batches. The few IOU points show the 20,000-batch prediction with the highest performance. However, because the validation data is not given for each batch, it is difficult to tell if the performance at 20,000 is indicative of a clearly better model. Future work includes changing the implementation of the model and training pipeline such that this can be evaluated.

To provide further insight into our model's performance, we ran real-time object detection with the duckiebot driving around the in-lab Duckietown environment. These brand-new images were clearly never seen by the duckiebot in training. Figure 4 shows a screenshot from these test images using model weights from various stages in training: 500 batches, 900 batches, 10,000 batches, and 20,000 batches. A YouTube video comparing the model predictions over many frames was also created.[6] The following section provides analysis for these results.

[6] `https://www.youtube.com/watch?v=Lyq2RXXD3mE&t=47s`

4. Discussion

The training IOU values as seen in Figure 3 appear quite noisy. Nevertheless, there is a clear upward trend as the training process continues, although the rate of increase becomes smaller after about 10,000 batches. As mentioned above, the YOLO training GitHub repository used did not provide functionality to evaluate validation metrics after each batch. It only saved the weights after certain batch numbers. These saved weights were used to evaluate the validation IOU, included in the plot.

Our testing trials on the duckiebot in real-time (one frame showed in Figure 4) demonstrated the 20,000-batch model's ability to recognize most objects close to the duckiebot. As can be seen, the detections from 500 and 900 batches into training do not recognize the duckiebot, whereas the later ones do. Additionally, the 20,000-batch model has similar bounding box predictions as the 10,000-batch model, yet outputs higher confidence scores. Thus, we can see that choosing the 20,000-batch model outputs bounding box detections for close objects in the image with higher confidence than previous models.

When displaying predictions, the threshold for confidence was set to 0.3. This may seem quite low (the model is only 30% confident that the bounding box contains the object), yet worked in our case because there were little to no false positives. However, there were cases where the detection failed. Duckiebots were usually able to be detected from a side view, yet were detected much less often when viewing from the back (demonstrated in Figure 5a). Another failure case, shown in Figure 5b, occurred when the duckiebot was partially occluded and very close to the camera. It is likely that these failures occurred due to these duckiebot orientations not appearing often enough in the training set. With more data representing these orientations, the detections would likely improve.

The YOLO model was initially ran on the duckiebot itself, although the limited processing power on the Raspberry Pi achieved an inference time of only 0.14 frames per second (meaning that each frame would take about 7 seconds to make predictions) - too slow for a real-time driving application. We considered using a specialized lightweight object detection network called (Wu et al., 2017a). However, the SqueezeDet implementation on GitHub[7] was specific to the KITTI dataset and would need nontrivial refining for our application. Because it was announced that the use of a GPU server would be available during the AIDO competition, we decided that the YOLO model had a fast enough inference time using a GPU for real-time (30 FPS) predictions and continued to refine this model.

[7] `https://github.com/BichenWuUCB/squeezeDet`

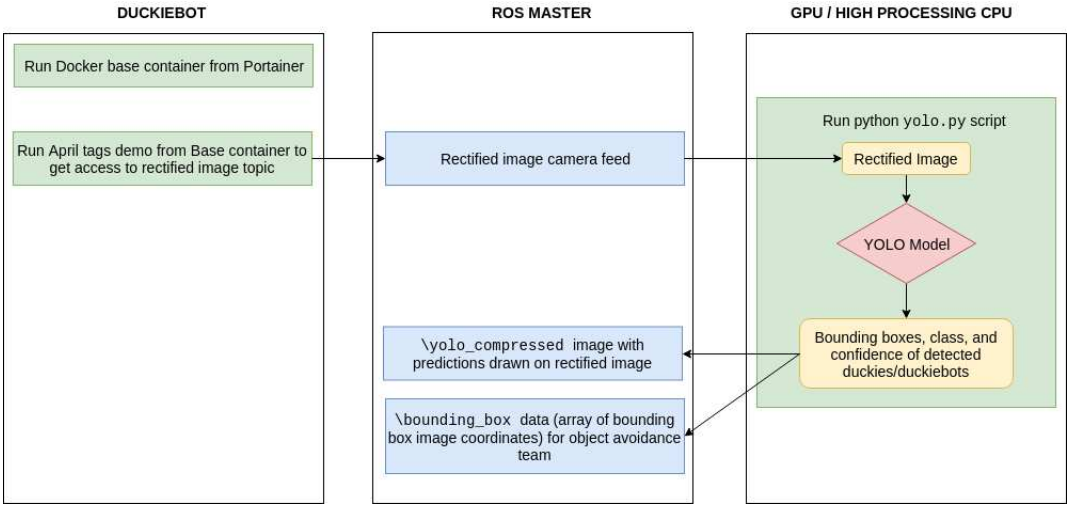


Figure 2. Flowchart demonstrating interfacing between code running on the duckiebot and GPU/CPU, communicating via ROS.

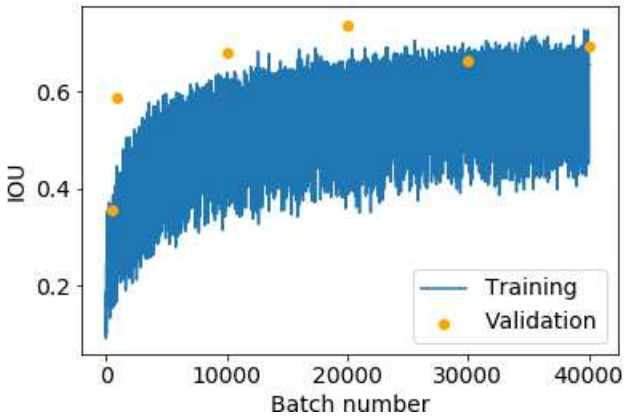
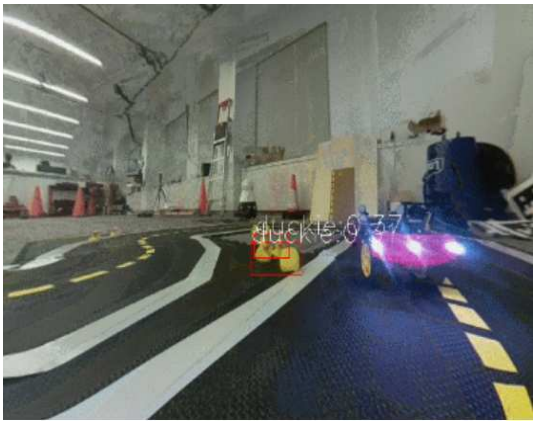


Figure 3. Training and Validation IOU over model training process.

(a) Detection after 500 training batches.

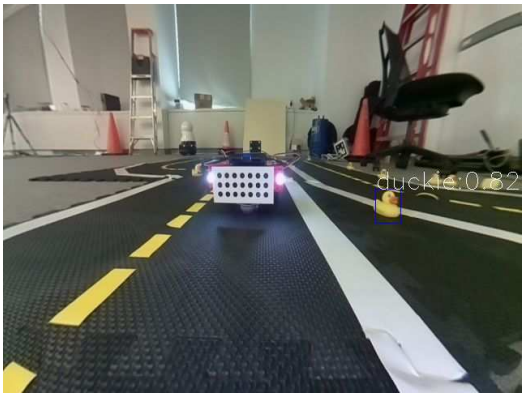


(b) Detection after 900 training batches.

(c) Detection after 10,000 training batches.

(d) Detection after 20,000 training batches.

Figure 4. Detections on test image with YOLO model at various numbers of batches into training process.



(a) Duckiebot is not detected from back view.

(b) Duckiebot is not detected when partially occluded and close to camera.

Figure 5. Sample failure cases of duckiebot detection.

5. Future Work

One improvement for future work on this project would be to find an implementation of YOLO on a more popular machine learning framework, such as TensorFlow or PyTorch, or implement it by hand. The repository used in this work, despite being forked from the original YOLO GitHub, was poorly documented and not well suited for custom extensions. Even the relatively common practice of validating a model during training was not offered. Finding or creating a different implementation would be very valuable for further work.

As discussed in Section 4, there were several failure cases where duckiebots were not detected. This could be improved with the addition of more training data including duckiebots at various orientations. Additionally, all the duckiebots in the training set had the white LED lights on. This creates a dataset bias in which duckiebots with different LED lights (or none at all) would likely be difficult to detect. The dataset was also constructed with images only from the Cornell Autonomous Systems Lab Duckietown environment. Collecting training data from other Duckietowns would likely improve generalizability. Another method would be to train in a manner similar to the Falling Things synthetic dataset for 3D object detection (Tremblay et al., 2018a). The effectiveness of this method for robotic grasping has been demonstrated (Tremblay et al., 2018b), although it required an intensive data collection process of 3D objects even before passing them to a dataset synthesizer. Future work on the project described in this paper may include simplifying the synthetic dataset such that the input would only require the 3D meshes of duckies and duckiebots found in the official Duckietown repository.

6. Imitation Learning

6.1. Prior Work

In the Fall 2018 semester, I worked with Jahanvi Kolte on imitation learning for Duckietown. Imitation learning is a machine learning process where the model input is an image from the duckiebot camera, and the model output is the 2 values defining the left and right angular wheel velocities. Thus, one model goes from perception directly to control. The final report can be accessed through a shareable Google Drive link [8]. This report details the project setup, data collection, and data preprocessing. The focus of the prior work was to compare lightweight neural networks using shift operations (Wu et al., 2017b) to their convolutional counterparts on the imitation learning task.

[8] https://tinyurl.com/AIDO-imitation-learning

6.2. Recent Extensions

This semester I discovered a bug in the loss calculation from prior work, in which we had divided the loss by the number of samples in the batch, although it had already been averaged. This resulted in much smaller mean squared error than in reality. The results in this report are calculated correctly.

Additionally, it was difficult last semester to download video logs of duckiebot driving for training data. The server hosting these videos was unresponsive for file downloads. As a result, only 5 videos were able to be downloaded. Earlier this spring, I revisited the server and was pleasantly surprised to find that the server was running again. I downloaded more training videos, increasing the data available from 4,154 frames to 14,098 frames. The training set had frames from 8 videos, whereas 5 videos were reserved for the test set. Sample frames from each of these videos are shown in Figures 8 and 9. These figures provide insight into the variability between training and testing data used here.

The goal of this work was simply to train a model to generalize to unseen data, rather than last semester's goal of lightweight network comparisons. I experimented with training a ResNet20 and ResNet56 from scratch, in addition to fine-tuning a ResNet18 pretrained on ImageNet. The GitHub repository containing my work can be found at `https://github.com/spetryk/imitation-learning-AI-Driving-Olympics`.

6.3. Results

6.3.1. Training from scratch

I began by training a ResNet20 from scratch without normalizing the input images, results of which are given in Figure 6a. I then found the training dataset channel-wise means and standard deviations, and trained another ResNet20 with normalized inputs for comparison (Figure 6b). The normalization may have helped to make the inputs from various Duckietown environments have less variability. Next, I was curious as to how a larger model would perform, and trained a ResNet56 from scratch (Figure 6c). All models were trained for only 30 epochs as an exploration into initial performance.

6.3.2. Fine-tuning pretrained network

I then explored the use of a Resnet18 pretrained on ImageNet. The pretrained model weights and architecture were available through the PyTorch framework.[9] The first model trained used a learning rate of 0.01 as initialization to an Adam optimizer, which was the same setting used for the models in Section 6.3.1. This result is shown in Figure 7b. Because the validation loss had plateaued, I wondered if a lower learning rate may help avoid a potential local minimum by taking a different path through the loss surface during training. Figure 7a shows results with an initial learning rate of 0.001.

[9] `https://pytorch.org/docs/stable/torchvision/models.html`

6.4. Discussion

The normalization for the Resnet20 models helped to stabilize the training, as the training loss converged much faster, and the validation loss was less variable. Thus, normalization was used for the rest of the models trained. The larger model, Resnet56, took longer to stabilize error. However, the training was stopped at 30 epochs, before the training loss could converge. These initial experiments were explorations into possible models, and therefore the models were not trained for an extensive amount of time. It would be valuable to see the performance with longer training times.

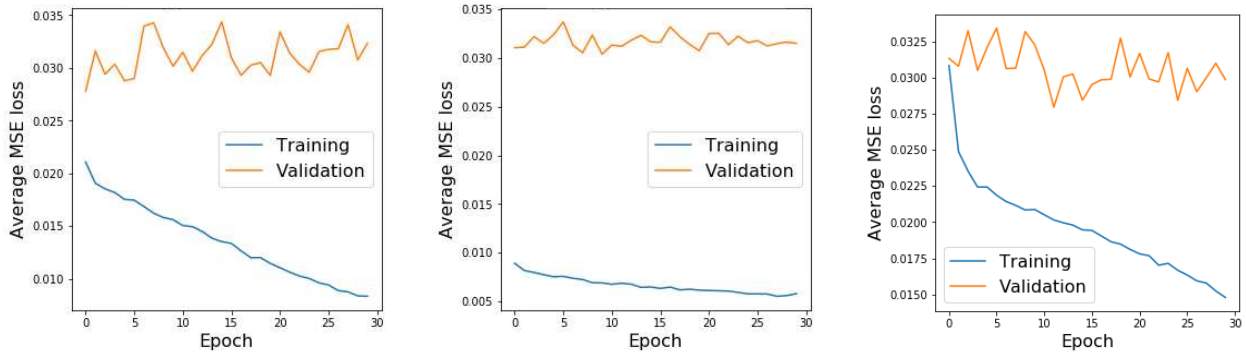
Surprisingly, the pretrained models did not perform significantly better throughout the fine-tuning process than those models trained from scratch. The training and validation accuracies appeared to plateau at similar values: training loss around 0.007 and validation around 0.030. The lower learning rate led to noisier validation loss, although did succeed in achieving a slightly lower validation loss.

In all models, the validation loss plateaued around 0.03 $(rad/s)^2$. Compared to the standard deviation of wheel velocities over the training set, about 0.20 rad/s (or 0.04 $(rad/s)^2$), this seems high. The training loss, on the other hand, appeared to be steadily decreasing for all models, reaching losses below 0.01 $(rad/s)^2$, low compared to the dataset standard deviation. This result is promising for autonomous driving in similar environments.

The test data was comprised of 5 driving videos separate from the 8 driving videos in the training set (there were no frames in the test and train sets that came from the same video). Figure 8 shows one frame from each of 8 training videos, and Figure 9 shows one frame from each of 5 test videos. A noticeable difference in environments can be observed. This makes the generalization of imitation learning more difficult. This difficulty is evident in the high validation loss.

6.5. Future Work

While the models appeared to struggle with generalization to unseen environments, more rigorous training may still have the potential to decrease this error. This may include the further tuning of hyperparameters, change of models (here, only Resnet was explored, but other CNN architectures such as DenseNet or a custom network may perform differently), and training for more iterations. Additionally,

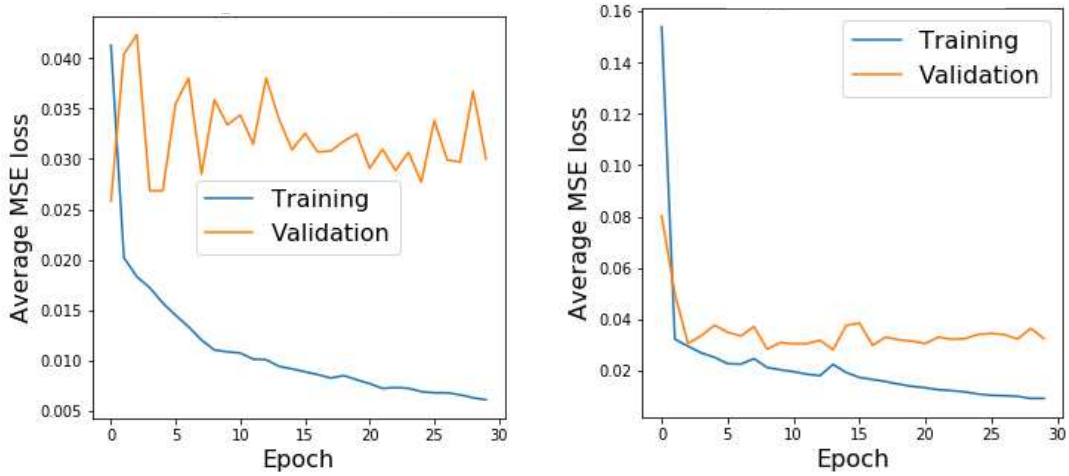


(a) Resnet20 training from scratch without normalization of input images.

(b) Resnet20 training from scratch with normalization of inputs.

(c) Resnet56 training from scratch with normalization of inputs.

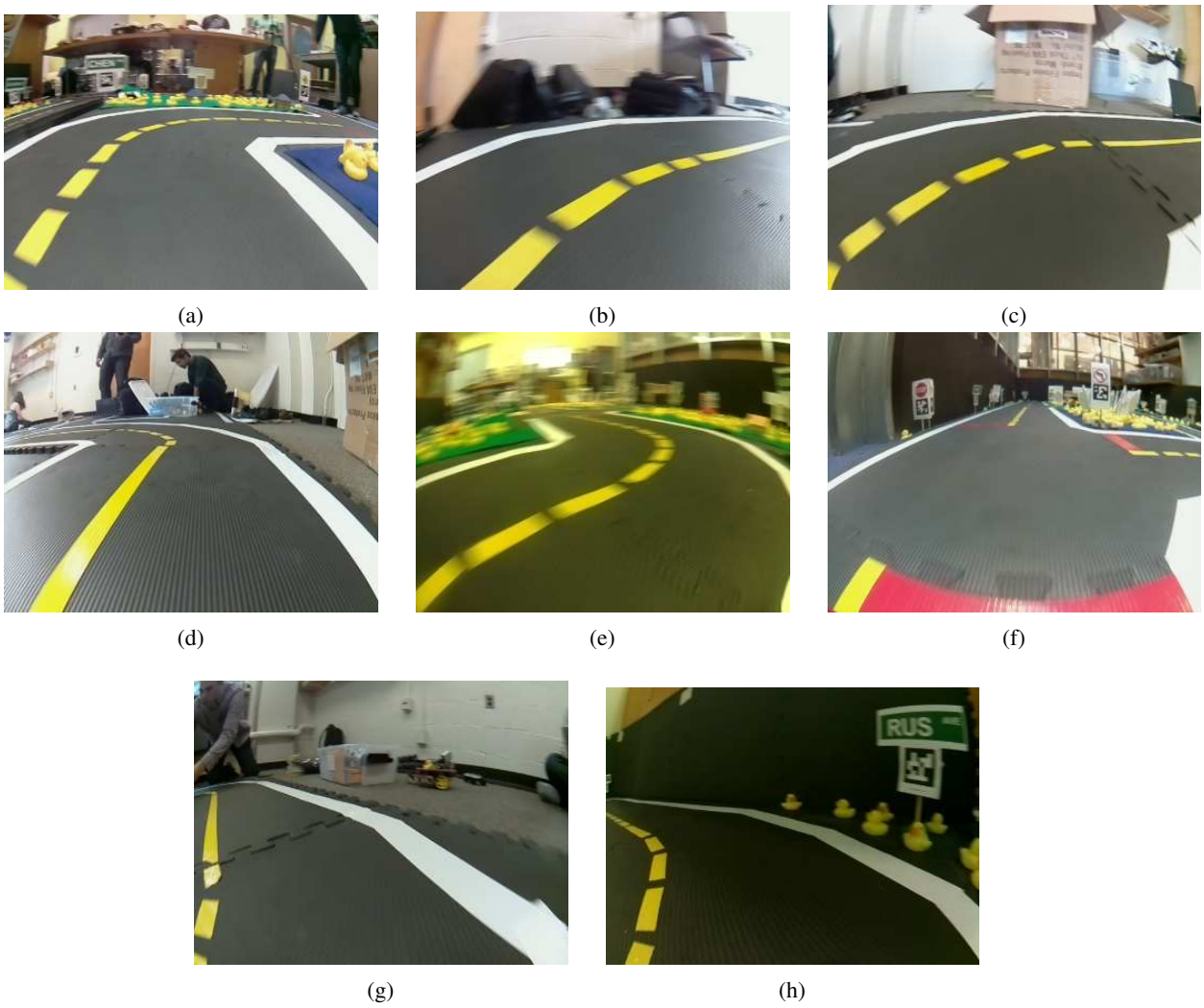
Figure 6. Results from training Resnet20 and Resnet56 from scratch.



(a) Pretrained Resnet18 fine-tuning with initial learning rate of 0.001.

(b) Pretrained Resnet18 fine-tuning with initial learning rate of 0.01.

Figure 7. Results from fine-tuning pretrained Resnet18, varying the learning rate.



(a) (b) (c)

(d) (e) (f)

(g) (h)

Figure 8. Sample frames from 8 different training set videos.

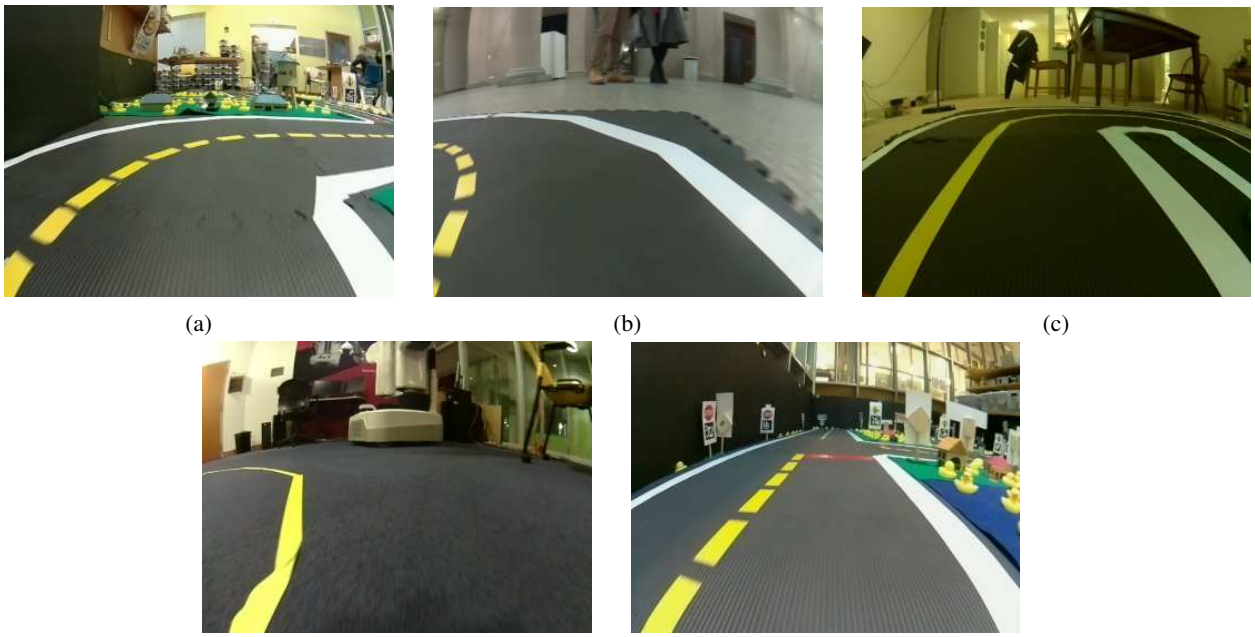


Figure 9. Sample frames from 5 different test set videos.

the ability of the models to reach low training loss after only 30 epochs is very promising for driving in familiar environments. Training data can be collected by driving a duckiebot around with keyboard control in the Duckietown in the Cornell Autonomous Systems Lab. Then, the test distribution will be exceedingly similar to training, since the environment is the same. This has the potential to greatly help generalization. After this training, experimentation and control on a physical duckiebot is a promising next step.

7. Real-World Autonomous Driving Reflection

This project made me better understand the difficulties facing object detection for real-world autonomous driving. Even in this very regular Duckietown environment where the appearance of the objects was constant, there were still failure cases of object orientations that were not detected. With infinitely more variability in a real-world setting, detectors must be incredibly robust for reliability in potentially life-threatening situations. I am curious on the implementation of object detection on actual autonomous vehicles, specifically the level of sophistication beyond YOLO and the importance given to running with low power and fast inference times.

Acknowledgements

I would like to acknowledge Brian Wang and Vikram Shree for their insight on this object detection project, as well as Scott Hamill, Adam Pacheck, and the other members of the Cornell AIDO team for their involvement in the Duckietown project at the Cornell Autonomous Systems Lab. I would also like to thank Prof. Hadas Kress-Gazit and Prof. Mark Campbell for their support of the Duckietown project at Cornell.

References

Dalal, Navneet and Triggs, Bill. Histograms of oriented gradients for human detection. In *international Conference on computer vision & Pattern Recognition (CVPR'05)*, volume 1, pp. 886–893. IEEE Computer Society, 2005.

Girshick, Ross. Fast r-cnn. In *The IEEE International Conference on Computer Vision (ICCV)*, December 2015.

Girshick, Ross, Donahue, Jeff, Darrell, Trevor, and Malik, Jitendra. Rich feature hierarchies for accurate object detection and semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 580–587, 2014.

Liam Paull, Jacopo Tani, Heejin Ahn Javier Alonso-Mora Luca Carlone Michal Cap Yu Fan Chen Changhyun Choi Jeff Dusek Yajun Fang and others. Duckietown: an open,

inexpensive and flexible platform for autonomy education and research. pp. 14971504, 2017.

Lowe, David G. Distinctive image features from scale-invariant keypoints. *International journal of computer vision*, 60(2):91–110, 2004.

Quigley, Morgan, Conley, Ken, Gerkey, Brian, Faust, Josh, Foote, Tully, Leibs, Jeremy, Wheeler, Rob, and Ng, Andrew Y. Ros: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, pp. 5. Kobe, Japan, 2009.

Redmon, Joseph, Divvala, Santosh, Girshick, Ross, and Farhadi, Ali. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 779–788, 2016.

Ren, Shaoqing, He, Kaiming, Girshick, Ross, and Sun, Jian. Faster r-cnn: Towards real-time object detection with region proposal networks. In Cortes, C., Lawrence, N. D., Lee, D. D., Sugiyama, M., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems 28*, pp. 91–99. Curran Associates, Inc., 2015.

Tremblay, Jonathan, To, Thang, and Birchfield, Stan. Falling things: A synthetic dataset for 3d object detection and pose estimation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pp. 2038–2041, 2018a.

Tremblay, Jonathan, To, Thang, Sundaralingam, Balakumar, Xiang, Yu, Fox, Dieter, and Birchfield, Stan. Deep object pose estimation for semantic robotic grasping of household objects. *arXiv preprint arXiv:1809.10790*, 2018b.

Tzutalin. Labelimg, 2015. URL https://github.com/tzutalin/labelImg.

Wu, Bichen, Iandola, Forrest, Jin, Peter H, and Keutzer, Kurt. Squeezedet: Unified, small, low power fully convolutional neural networks for real-time object detection for autonomous driving. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pp. 129–137, 2017a.

Wu, Bichen, Wan, Alvin, Yue, Xiangyu, Jin, Peter, Zhao, Sicheng, Golmant, Noah, Gholaminejad, Amir, Gonzalez, Joseph, and Keutzer, Kurt. Shift: A zero flop, zero parameter alternative to spatial convolutions. *arXiv preprint arXiv:1711.08141*, 2017b.