Learning end-to-end control for duckiebot driving

Group: Dropouts

June 2021

1 Group Information

Members:

- Keyan Pishdadian (keyanp@cs.washington.edu)
- Jakub Filipek (balbok@cs.washington.edu)
- Kyle Deeds (kdeeds@cs.washington.edu)

Source code for behavior cloning approach:
`https://github.com/keyan/duckiebot_behavior_cloning`
Source code for reinforcement learning approach:
`https://github.com/balbok0/cse571-sp21-project-2-dropouts`

2 Task Overview

In this project we sought to explore methods for learning an end-to-end control policy that would allow the duckiebot to navigate tracks autonomously. Our goal was to enable the duckiebot and have it follow a lane and navigate around an entire track without any major driving infractions (defined as exiting the lane or going off the road), using only monocular camera data. Ideally this system would be robust enough to generalize to new tracks and work on the real robot as well as in simulation.

Inspired by previous efforts in learning end-to-end control for driving using deep reinforcement learning [KHJ+18] our original plan was to evaluate a selection of reinforcement learning algorithms to solve this task. Knowing that reproducing reinforcement results is difficult and often unsuccessful [HIB+19], we planned to also experiment with a behavior cloning approach inspired by prior work from NVIDIA where a convolutional neural network was trained to steer a car using only image data [BTD+16].

3 Environment Setup

We performed experimentation and evaluation of our agents in simulation and on the real robot primarily relying on the "AI Driving Olympics" infrastructure [ZTC+19]. Our agent was structured into the AIDO submission format and then executed either in simulation or on the real duckiebot. We used the default episode length configuration, which runs agents for 60 seconds. The AIDO submission format requires structuring an agent that obeys the AIDO agent interface and accepts image observations and outputs control commands. We referenced the baseline implementations for both reinforcement learning and behavior cloning submission types, `https://github.com/duckietown/challenge-aido_LF-baseline-RL-sim-pytorch` and `https://github.com/duckietown/challenge-aido_LF-baseline-behavior-cloning`, respectively.

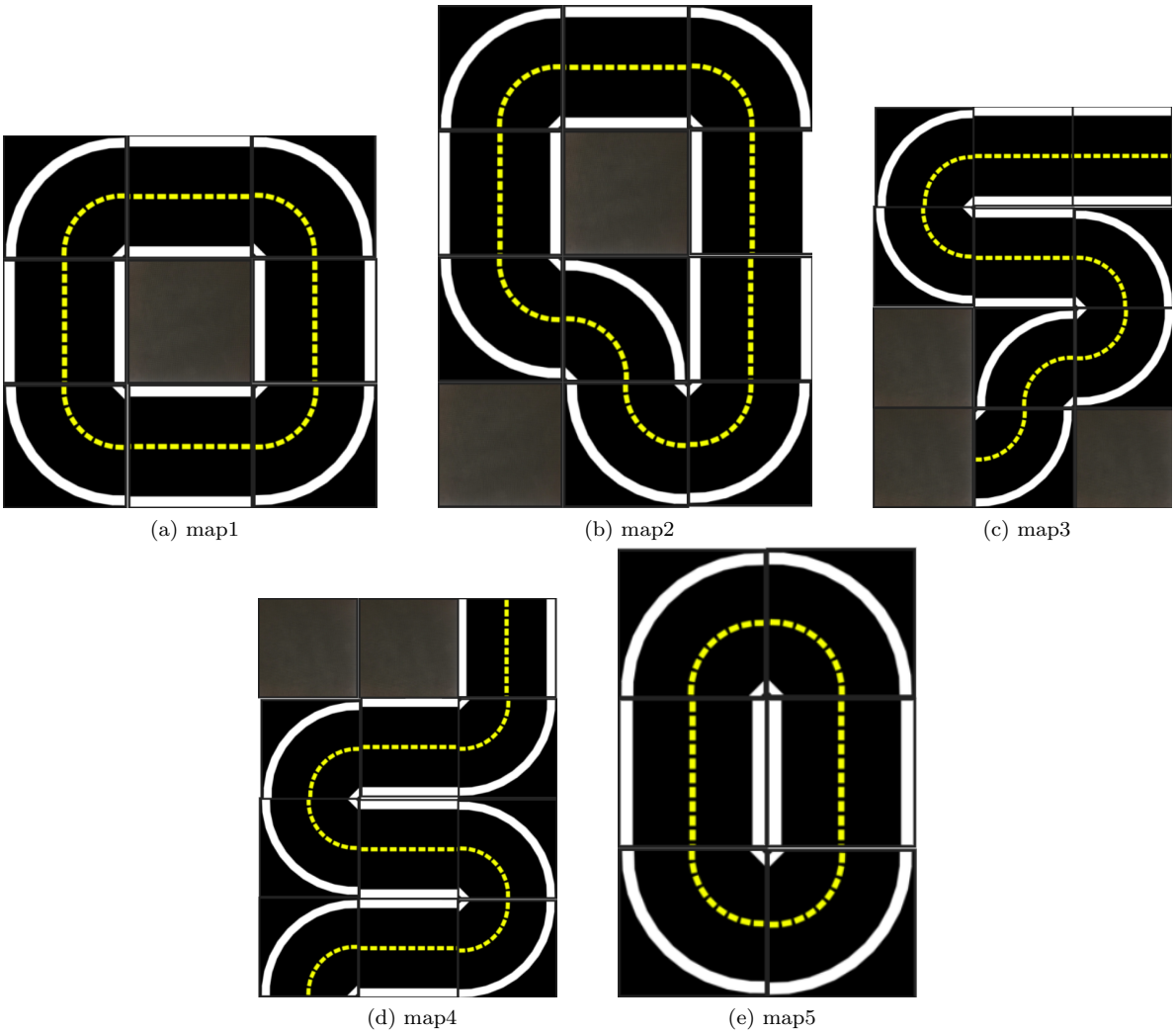
3.1 Simulation

We used the `gym-duckietown` simulator [CBGC+18] for training and evaluation of agents when using a reinforcement learning approach, as well as for evaluation when using behavior cloning (through the AIDO

submission infrastructure). For reinforcement learning we only ran evaluation on the most simple map type, "loop_empty", which is an square map with no obstacles. Evaluations for behavior cloning used a specific "challenge", `aido-LF-sim-validation` which runs the agent on a static selection of maps and ranks submissions against other users.

3.2 Real World

Evaluation on the real duckiebot was done on five different maps (1), with a 60 second maximum episode duration. Only map2 was used for training data collection.



(a) map1 (b) map2 (c) map3

(d) map4 (e) map5

Figure 1: Digital overview of the five maps used for real-world evaluation.

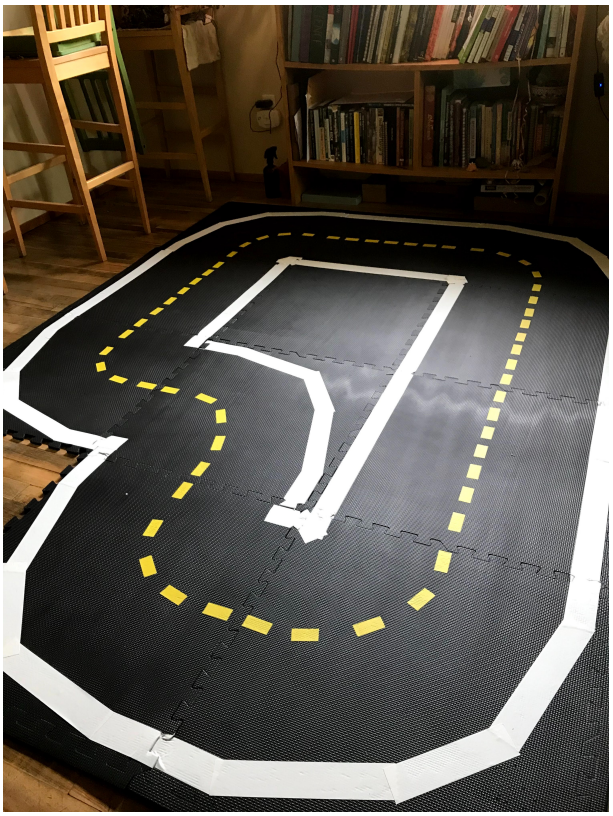


Figure 2: Photo of physical layout of map2.

4 Reinforcement Learning Approach

Our initial solution was to use the `gym-duckietown` environment to train an agent using a model-free off-policy Q-learning algorithm, for which we explored both DQN [MKS+13] and DDPG [LHP+16] algorithms. Our training code leveraged pre-written implementations of both algorithms provided through the RLLib package [LLN+17]. We leveraged the existing default reward function provided by the gym environment (`https://git.io/JGnsg`) which is a linear function of the robot speed, position within the lane relative to the right line, and proximity to obstacles.

Our first approach was DQN. In this case actor is learning the Q function (value function of state, action pair). In our case the network is given an image from the camera and is predicting the value over 3 possible actions, driving 45° to the left/right, and driving forward. During training DQN averages the reward function over the whole episode. These episodes are then played and collected into *replay buffer*, which when full triggers an iteration of neural network training.

Given enough training samples (our longest run was > 200 iterations, around 30 episodes each), the model should converge to a good result. Our main motivation for this network was that small set of discrete actions should overfit less to the simulation environment (this is because with slight changes to exact values of Q function calculated the maximal action would stay the same). However it has the obvious drawback of being much less flexible in its actions and possible recovery.

Our second approach was DDPG, which by working with continuous space addresses the main drawback of DQN, as stated above, while being extremely similar. The main two differences lie in the fact that DDPG uses actor-critic framework, where neural network outputs both the action and the predicted value function of state input and action tuple, as well as that the fact that weights of the network are using "soft" target, which causes them to change less and stabilizes learning.

Unfortunately we were unsuccessful in achieving reasonable training results with these methods, with our agent consistently showing an inability to increase average reward obtained despite experimenting with

domain randomization in the simulator, training across multiple maps, and performing hyperparameter search (Figure 3). Our initial impression was that training an agent to perform well in simulation would be straightforward and the main challenge would be in transferring the agent to the real environment. We tried customizing reward function to additionally benefit from distance travelled, but to no fruition. When our agents failed to show any promising results during training and subsequently failed to perform in simulation at all, we decided to abandon this approach. Videos of evaluation of these agents in simulation can be found at:

- DQN agent: `https://youtu.be/SDZ59_2zhGg`
- DDPG agent: `https://youtu.be/1xL8b2MK6N4`

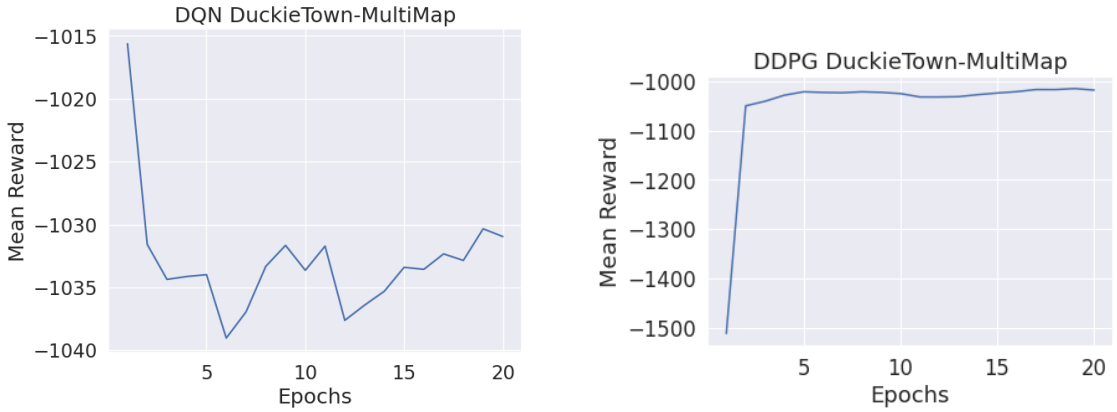


Figure 3: Mean reward obtained per epoch during training of the DQN (a) and DDPG (b) solutions. Both solutions failed to improve mean reward. Here only 20 epochs are shown, but the plateau in reward obtained was observed even when trained for many more epochs.

5 Behavior Cloning Approach

Our focus then shifted towards a strategy where we would train a convolutional neural network (CNN) that would accept a single camera image from the robot at each timestep and then output a prediction for the linear and angular velocity that the robot should apply. This approach was influenced by work from NVIDIA where this technique was applied to predicting a single value representing the steering of the vehicle [BTD+16]. We designed several CNN models (discussed in Section 5.2) and trained them on a combination of simulation and real world data, then evaluated these models using the AIDO leaderboard and on the real duckiebot. An high-level overview of this system is shown in Figure 4.

5.1 Data Preparation

Data collection from simulation in the `gym-duckietown` environment is easier, faster, and less noisy than real world data, so the overall data strategy was to focus on collecting a large volume of simulated data and augmenting its applicability to real robot evaluation by including a smaller amount of real robot data extracted from ROS bags.

To collect simulated data, a built-in demo was executed in the simulator that uses a pure pursuit controller [PCY+16] to geometrically determine the optimal robot controls, the resulting image observations and joystick controls were aggregated. Image observations from simulation were downsampled to 150x200 and domain randomization was applied during the simulation runs to help prevent overfitting to the simulation environment.

Real robot data was collected through a combination of accessing publicly available ROS bags [PTA+17] and ROS bags recorded from demonstrations with one of our own robots on map2 (Figure 1b). When selecting public robot logs, we attempted to find executions which were successfully navigating the map,

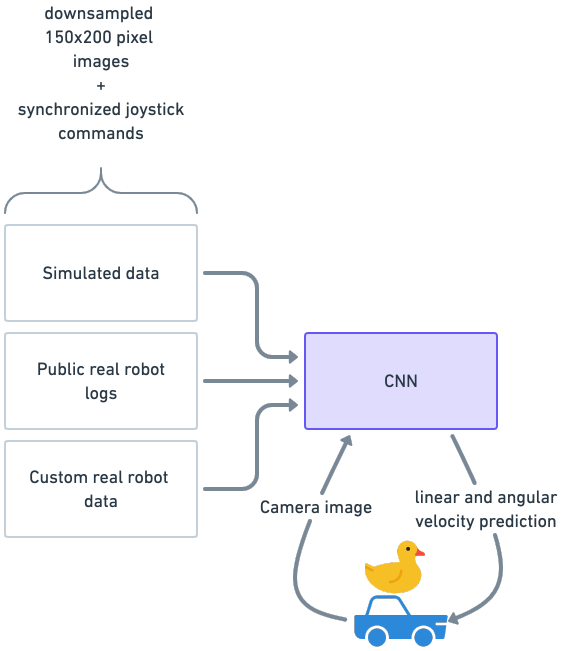


Figure 4: Overview of input training data and data flow during inference for the behavior cloning approach to end-to-end control. At runtime the robot passes each observation frame to the CNN model to produce predicted linear and angular velocities. These velocities are converted to left and right wheel velocity commands which are issued to the robot.

(a) Simulated data (b) Public log data (c) Custom data

Figure 5: Example frames from the three different data sources used for training. Observations are down-sampled 150x200 pixel images.

were free of obstacles, and contained turns. For the custom data robot controls were given by a human driver.

Ultimately we used two final datatsets. The largest (henceforth "dataset A") was used for training the initial model and consisted of 58,414 total observation frames with associated controls, this was a mix of simulated data collected from `gym-duckietown` (50,337 frames) and public real robot data (8,077 frames). The second dataset (henceforth "dataset B") was used for experimenting with fine tuning our models and consisted of 15,560 total frames, this only contained our custom real robot data.

5.2 Model Architectures

Our experiments were focused around two model architectures, the first was an attempt to reproduce results from NVIDIA [BTD+16] which we named the "nvidia" model (Figure 6a). In order to predict two values (linear and angular velocity) two separate instantiations of this model were created with disjoint weights. The only major deviations from the originally published model were the use of leaky rectified linear unit (LeakyReLU) activations and dropout in-between linear layers ($p = 0.5$). The second architecture used a smaller number of total parameters, shared weights across the convolutional layers, and two outputs preceded by small disjoint linear layers (Figure 6b). We named the initial version of this model "modelv0", subsequent experimentation produced a second version "modelv1" which substituted ReLU activations for LeakyReLU, and added dropout to the final linear layers ($p = 0.5$) and the shared convolutional layers ($p = 0.1$).

5.3 Training

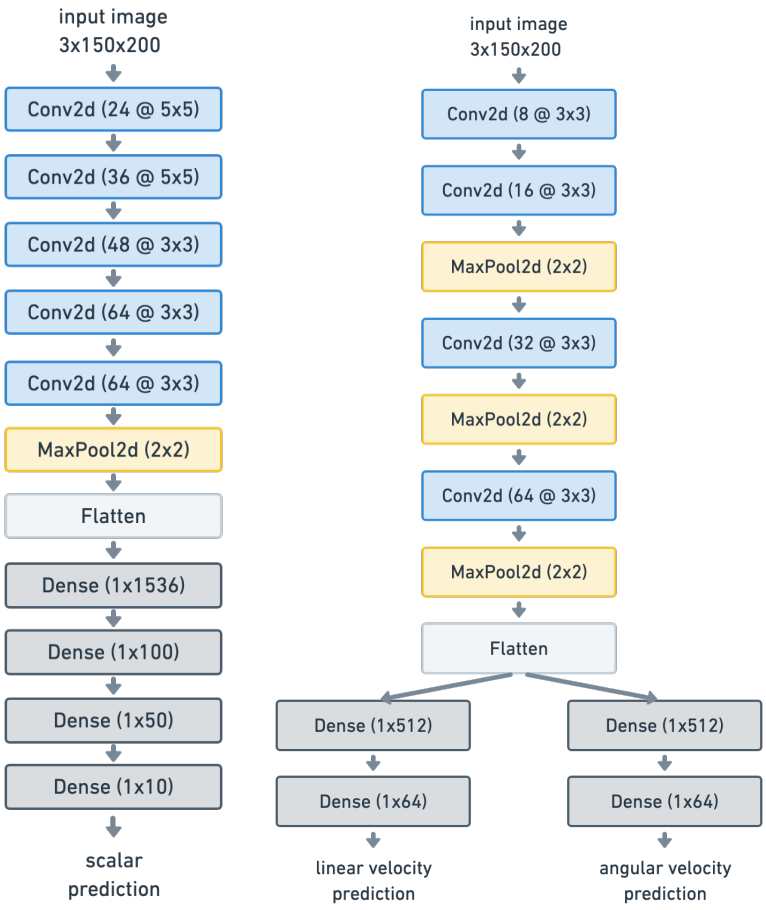
All training was done using stochastic gradient descent and mean squared error loss. The models were trained initially on the aggregated dataset ("dataset A") for 200 epochs, results in Figure 7. As the "nvidia" model did not show promising results during training, it was excluded from further experimentation. Then "modelv0" and "modelv1" were fine tuned for 50 epochs using the custom real robot dataset ("dataset B"), results in Figure 8. A prediction was deemed accurate if the continuous velocity value was within 5% of the label value.

6 Performance Evaluations

6.1 Simulator and AIDO

By writing an agent using the agent interface in the AIDO template, we were able to leverage the existing AIDO infrastructure for evaluating in simulation. We submitted all three of our models along with the baseline solution provided by Duckietown to the `aido-LF-sim-validation` challenge. This evaluates each solution across multiple runs and different maps, providing videos of robot execution in the simulator as well as overhead views of the robot trajectories. Visualization of the trajectories taken for all these models across three maps is shown in Figure 9. We noticed that our models performed roughly equivalent to the baseline solution or did worse.

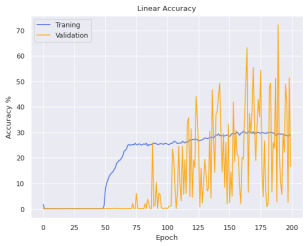
Following additional experimentation it appeared that the high linear velocity predicted by our models did not allow for sufficient time to make turns, and in general our models had difficulties in applying enough

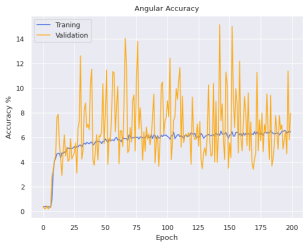


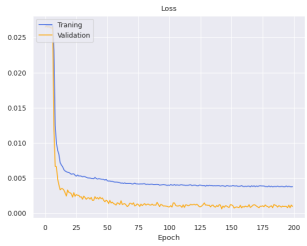
(a) (b)

Figure 6: Overview of the two primary model architectures evaluated in this work. (a) features a larger number of total parameters and no shared weights, each of the two prediction values retain their own copy of the model (b) uses shared convolutional layers and small linear layers preceding each of the two outputs.

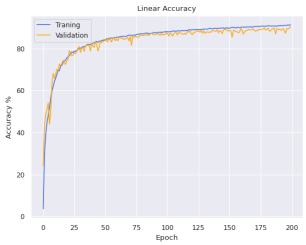
nvidia

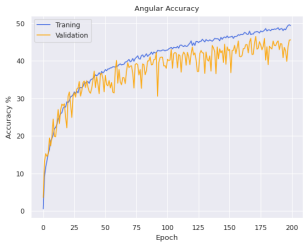


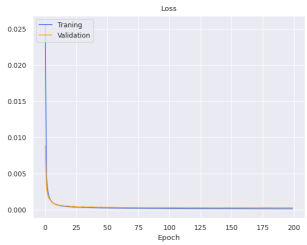




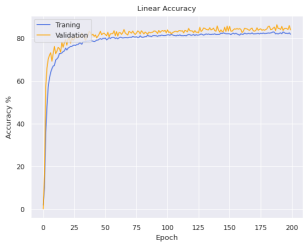
modelv0

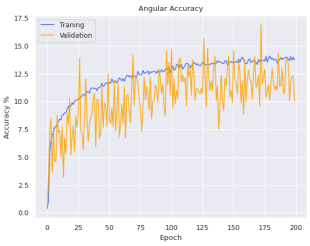






modelv1





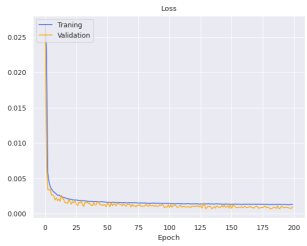
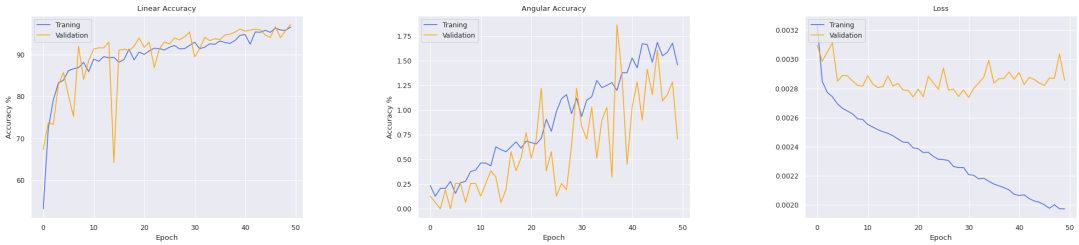


Figure 7: Results from training the three different model architectures on dataset A, showing linear accuracy, angular accuracy, and loss achieved. A prediction was considered correct if the continuous velocity value was within 5% of the label value.

modelv0



modelv1

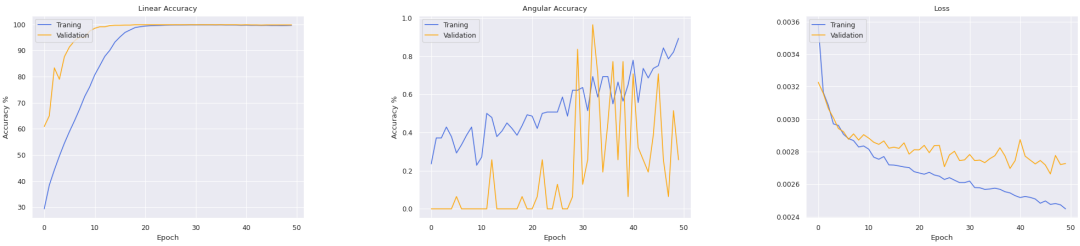


Figure 8: Results from fine tuning the shared weights models on dataset B, showing linear accuracy, angular accuracy, and loss achieved. A prediction was considered correct if the continuous velocity value was within 5% of the label value. Poor angular accuracy was achieved and training loss on the validation set plateaued quickly.

angular velocity to make turns effectively. To counteract this issue we applied hard coded modifications to the predicted linear and angular velocities at each command time step, dividing the predicted velocity by 2 and multiplying the predicted angular velocity by 1.5. Henceforth these manually modified models are signified as such with the suffix "*". This modification significantly improved the ability of our agent to navigate, which was our primary objective, even though it was at the cost of decreased travel distance within the 60 second episode length. We also applied this modification to the baseline solution in order to provide a better comparison, these results can be found in Figure 10.

Simulation results for the fine-tuned models were very poor, performing worse than the original unmodified models even. Visualizations of these results are not included for brevity, but real robot results are discussed below.

6.2 Real World

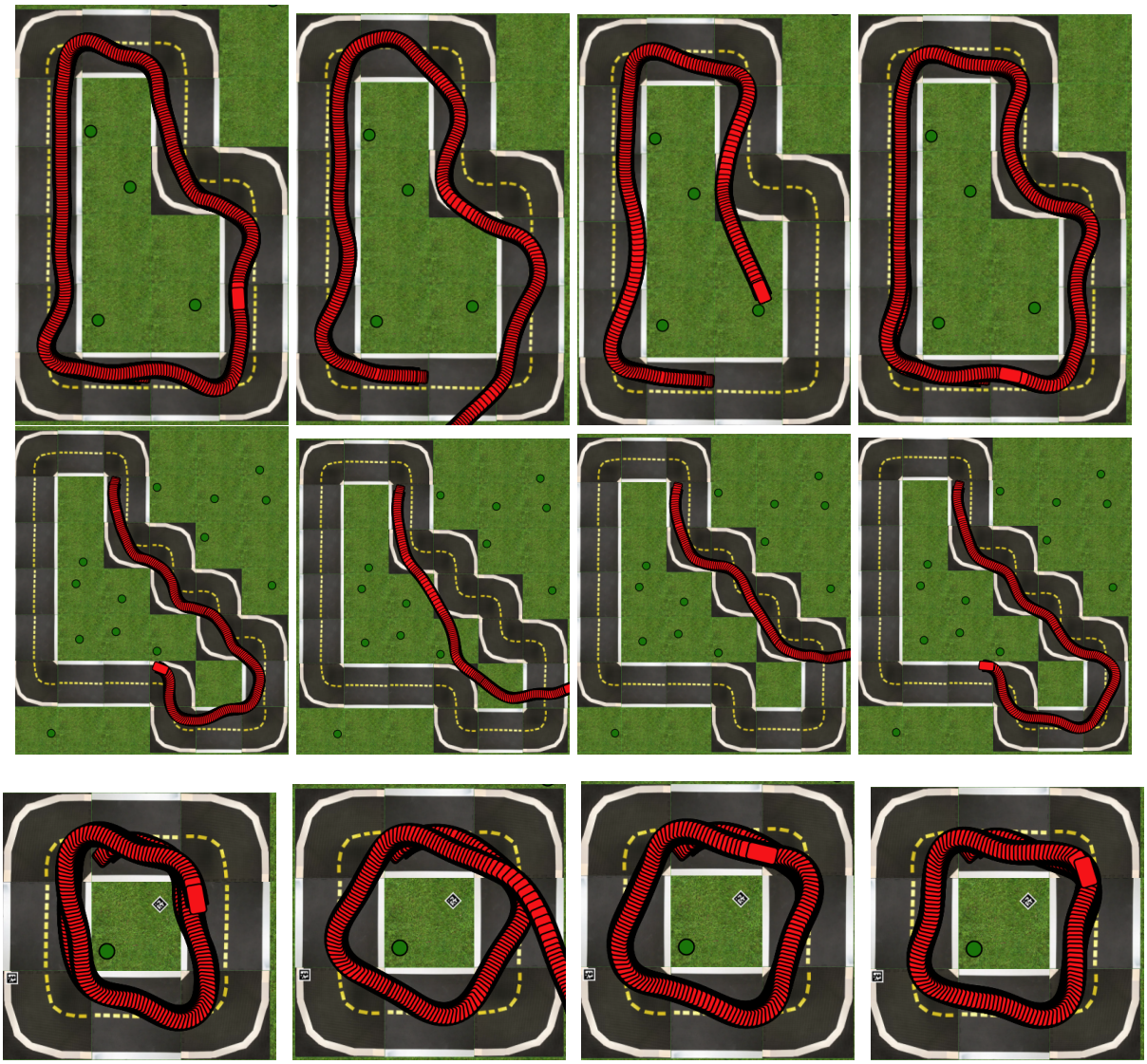
Having previously determined in simulation that the manually modified models ("nvidia*", "modelv0*", "modelv1*") performed best, we expected this result to be true during real robot evaluation as well. All initial evaluations were done on map2 (Figure 1b) and from these we determined that "modelv1*" was the overall best performing model, so it was evaluated on the other maps as well.

The following videos contain results from testing all three models along with the baseline solution on map2:

- original models: `https://youtu.be/j7rHbaK74h8`
- modified models: `https://youtu.be/6s80amO3GV8`

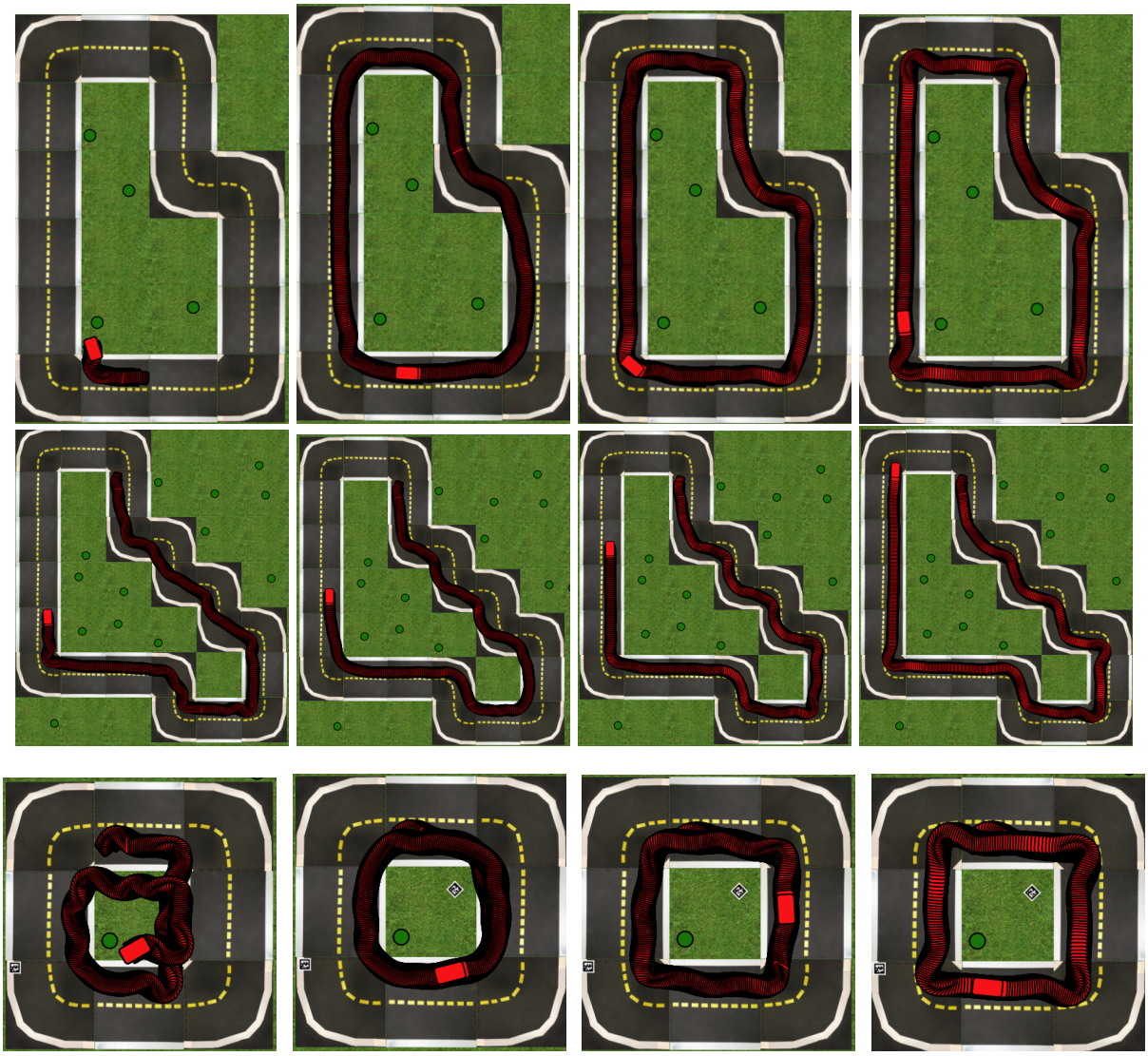
6.2.1 Fine Tuning

Consistent with the accuracy obtained during training and lack of ability to decrease validation set loss when performing fine tuning (Figure 8), real robot performance was much worse with this additional training, see `https://youtu.be/v5AOjATDAyU`. This is not altogether surprising though given that poor quality of the human driver controls used to create the custom dataset. It was simply too difficult to drive the robot even as a human given the coarse grained tools provided for user input and the overall quality and reliability of



(a) baseline (b) nvidia (c) modelv0 (d) modelv1

Figure 9: Robot trajectories in simulation on three different maps for the baseline solution (a) and each of the evaluated models (b-d).



(a) baseline* (b) nvidia* (c) modelv0* (d) modelv1*

Figure 10: Robot trajectories in simulation on three different maps for the baseline solution (a) and each of the evaluated models (b-d), in all cases the solutions were manually modified to apply a linear transformation of the predicted velocity values.

the robot. During human driving there were consistently moments where the robot seemed to drift off and execute some control the human did not apply. These factors likely contributed to a noisy dataset that did not improve the performance of our system.

6.2.2 Generalization

In order to evaluate the ability of our best performing model "modelv1*" to generalize to other maps it was tested alongside "baseline*" on the remaining maps (see below). In general the results were consistent with performance on map2 with the caveat that our solution could not handle right turns. This is likely an issue with our dataset having not enough examples of right turns and hence the robot was unable to use prior examples to successfully navigate this situation.

- map1: `https://youtu.be/jYgHov8mpX4`
- map3: `https://youtu.be/ALZrHWevRJ4`
- map4: `https://youtu.be/OodrOQcEKJM`
- map5: `https://youtu.be/zt69aAOWHX4`

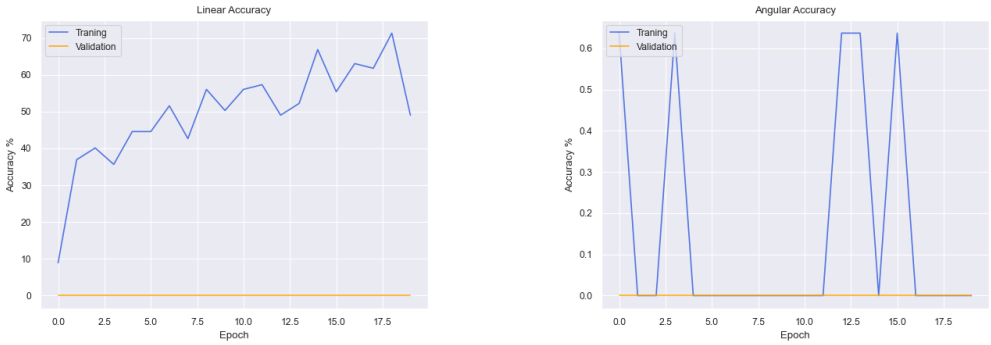
7 Potential Future Improvements

1. As explained in prior sections, one limitation faced was the difficultly in collecting good real robot data. Possibly one way to have circumvented this was to use the duckiebot's Xbox controller interface for human driving, assuming that this input possibly has better fine grained control.
2. As observed in our agent's inability to generalize to certain maps, we had a dearth of data with right hand turns. Improving our dataset to include more diverse situations would likely improve real world performance.
3. In general issues with the robot caused limitations in our groups ability to properly evaluate our agents. One group member's robot had consistent issues with keeping its wheel on and rendered him unable to use his robot in a meaningful way. The use of more robust robot components would certainly improve the ability to iterate during experimentation and also have a system less subject to control noise.
4. We suspected that performance at inference time might have been a bottleneck for our system. Possibly this is one reason that introducing a hard coded linear velocity reduction improved performance, by allowing time to do inference on observations corresponding to turn taking. A potential place to look for performance improvement would be to either quantize our PyTorch model to speed up CPU-based inference, or to adapted our system to use the onboard GPU on the Jetson Nano. Despite some effort in trying to adapt the duckietown software stack to use the GPU, the containerized software system was unable to detect the on-board GPU and hence we could not leverage it.
5. We also believe that using a recurrent network model has the potential to improve performance. Although our data input is actually video, we treated it as isolated image frames and lost the ability to leverage prior observations in making predictions.

7.1 3D Convolutions

To at least partially address the fifth potential improvement we decided to try 3D Convolution, where the additional dimension of kernel was over time.

Different approach requires different data processing. We first split processed each log separately into episodes. For sentence length $S = 128$, we pad the first frame S times to the beginning of each episode in training data. This is done to force robot to learn how to start driving. Then we iterate over these frames with step 10 creating videos of length S at each step. For each video we define the target angular and linear velocities based on the last frame. Thus each observation was $\mathbb{R}^{n \times 128 \times 3 \times 150 \times 200}$ (where n is batch size),



(a) Linear Accuracy

(b) Angular Accuracy

Figure 11: Results from training model with 3D convolutions.

while both velocities were $\mathbb{R}$. Lastly, since angular velocities are skewed significantly towards $0°$ training data is filtered so that there are equal amounts of examples with $|\omega| > 0.25$ and $|\omega| \leq 0.25$.

Model architecture uses multiple 3D convolution layers with decreasing number of channels (time dimension) from 128 to 16. Model then reshapes data to $\mathbb{R}^{n \times 48 \times 150 \times 200}$ and passes it through architecture similar to V1.

The results of the model were disappointing, as it failed to successfully predict almost any angular velocity correctly. Moreover it also overfitted to training data, while performing worse than all image based models.

In conclusion we believe that video-based inference is a viable approach, however 3D convolutions are not the best suited for the task, or are extremely difficult to find correct hyperparameters for.

References

[BTD+16] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D. Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, Xin Zhang, Jake Zhao, and Karol Zieba. End to end learning for self-driving cars. *CoRR*, abs/1604.07316, 2016.

[CBGC+18] Maxime Chevalier-Boisvert, Florian Golemo, Yanjun Cao, Bhairav Mehta, and Liam Paull. Duckietown environments for openai gym. `https://github.com/duckietown/gym-duckietown`, 2018.

[HIB+19] Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. Deep reinforcement learning that matters, 2019.

[KHJ+18] Alex Kendall, Jeffrey Hawke, David Janz, Przemyslaw Mazur, Daniele Reda, John-Mark Allen, Vinh-Dieu Lam, Alex Bewley, and Amar Shah. Learning to drive in a day. *CoRR*, abs/1807.00412, 2018.

[LHP+16] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. In Yoshua Bengio and Yann LeCun, editors, *ICLR*, 2016.

[LLN+17] Eric Liang, Richard Liaw, Robert Nishihara, Philipp Moritz, Roy Fox, Joseph Gonzalez, Ken Goldberg, and Ion Stoica. Ray rllib: A composable and scalable reinforcement learning library. *CoRR*, abs/1712.09381, 2017.

[MKS+13] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013.

[PCY+16] Brian Paden, Michal Cáp, Sze Zheng Yong, Dmitry S. Yershov, and Emilio Frazzoli. A survey of motion planning and control techniques for self-driving urban vehicles. *CoRR*, abs/1604.07446, 2016.

[PTA+17] Liam Paull, Jacopo Tani, Heejin Ahn, Javier Alonso-Mora, Luca Carlone, Michal Cap, Yu Fan Chen, Changhyun Choi, Jeff Dusek, Daniel Hoehener, Shih-Yuan Liu, Michael Novitzky, Igor Franzoni Okuyama, Jason Pazis, Guy Rosman, Valerio Varricchio, Hsueh-Cheng Wang, Dmitry Yershov, Hang Zhao, Michael Benjamin, Christopher Carr, Maria Zuber, Sertac Karaman, Emilio Frazzoli, Domitilla Del Vecchio, Daniela Rus, Jonathan How, John Leonard, and Andrea Censi. Duckietown: an open and inexpensive and flexible platform for autonomy education and research. In *IEEE International Conference on Robotics and Automation (ICRA)*, Singapore, May 2017.

[ZTC+19] Julian Zilly, Jacopo Tani, Breandan Considine, Bhairav Mehta, Andrea F. Daniele, Manfred Diaz, Gianmarco Bernasconi, Claudio Ruch, Jan Hakenberg, Florian Golemo, A. Kirsten Bowser, Matthew R. Walter, Ruslan Hristov, Sunil Mallya, Emilio Frazzoli, Andrea Censi, and Liam Paull. The ai driving olympics at neurips 2018. *arXiv preprint arXiv:1903.02503*, 2019.