Learning from Sub-optimal Data: Reducing Policy Shakiness in Visual Autonomous Driving with Imitation Learning

William Chen
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA, 02139
`verityw@mit.edu`

& **Alex Cuellar**
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA, 02139
`alexcuel@mit.edu`

Abstract

Deep learning control methods have found success in fields like autonomous driving. Such methods include both reinforcement learning and end-to-end imitation learning. Both of these algorithm classes are able to leverage collected expert policy data sets. However, with more complex tasks, expert performance cannot be ensured, and these systems will rely on sub-optimal example data, which may induce undesirable high frequency or shaky control noise. We thus explore the effects of various neural network architectures on self-driving policy shakiness in the imitation learning case, where a simulated car is steered by a neural network trained on limited non-expert human driving data. We explore various feed-forward and recurrent architectures, and evaluate the shakiness of their learned policies' output steering angles. We also test and evaluate a policy-smoothing filter as another solution to this problem. Finally, we present possible explanations for our results.

1 Introduction

Machine learning for robotics and automated control systems is fast becoming a viable technology. However, the results of such methods can produce results effective within the fundamental requirements of a problem, but lack in some more nuanced metrics required for fully-functioning systems. One of the most apparent short-comings of ML-based control systems is the often shaky behavior resulting from learned policies. In physical systems, the high-frequency noise in shaky policies cannot be carried out and can significantly disrupt performance.

Ultimately, this noise seen in policies is caused by learning frameworks that do not penalize such behavior – if a model is rewarded for simply achieving a task successfully, it will have no reason to avoid noisy policies. Therefore, in the project, we aim to reduce noise in learned robotic systems by altering standard machine learning frameworks to punish noisy policies.

Despite the fact that this problem is most discussed in Reinforcement Learning, due to hardware constraints, we will explore the problem of shaky policies in imitation learning. In order to achieve, this we will utilize suboptimal demonstrations that induce shaky policies. Then, we will test our proposed solutions against a baseline method with respect to the success rate of the resulting models and the reduction in high frequency noise.

2 Related Works

Much of the work that has been done on learning from sub-optimal demonstrations has been with regards to reinforcement learning, though some algorithms incorporate imitation learning into their structure as well.

The Trajectory-ranked Reward Extrapolation (T-REX) algorithm is designed to infer an underlying reward function from a collection of sub-optimal demonstrations [4]. Specifically, it assumes that certain trajectories are labelled as being preferable to others, and trains a neural network predicting the reward at a given state such that more preferred trajectories have higher scores than less preferred ones. This reward function can then be used for reinforcement learning.

The Disturbance-based Reward Extrapolation (D-REX) algorithm naturally extends upon the above by removing the need for preference labelling, as the procedure generates preferences automatically [5]. The algorithm first uses behavior cloning from the demonstrations to get a rough policy. Then, it injects varying amounts of noise into the policy, running it to collect additional trajectories. Lastly, it assigns preferences by giving higher rankings to policies with less noise, ultimately running T-REX with said rankings.

Lastly, the Self-Supervised Reward Regression (SSRR) algorithm improves upon D-REX by removing an inductive bias that the latter has [6]. SSRR improves upon reward estimates from inverse reinforcement learning methods like AIRL [7] by adding in varying amounts of noise to the trajectory collection policy and seeing the relation between reward and noise, fitting a sigmoid to the relation. Then, it trains a new neural network reward estimator to also consider said noise response, empirically improving many agents' performances in several key RL tasks.

3 Methodology

3.1 Problem Statement

We wish to train a neural network to imitate human driving behaviors and successfully steer a virtual car around a simulated track without driving off the road, crashing, or driving shakily. That is, the networks should take in camera view data and output steering commands. However, in this imitation learning framework, only highly sub-optimal training data will be available – other than in the control group test case, our neural networks will not have access to "desirable" human driving patterns around the track. We thus hope to investigate how different policy smoothing methods and neural network architectures would control the car when trained upon this data.

3.2 Simulator

We make use of Udacity's Self-Driving Car simulator [3]. This program has the following vital functionalities:

Training Mode

- *Manual Control*: A human operator can manually control the virtual car by using their keyboard's arrow keys or the WASD keys.
- *Automatic Data Recording*: While controlling the car manually, the Udacity simulator has built-in data recording functionality, allowing the user to record images (from front and two lateral simulated cameras on the car) along with corresponding telemetry information, like steering angle, speed, and acceleration.

Autonomous Mode

- *Websocket Support*: The simulator can be interfaced with via Python's SocketIO library, allowing for the following two points.

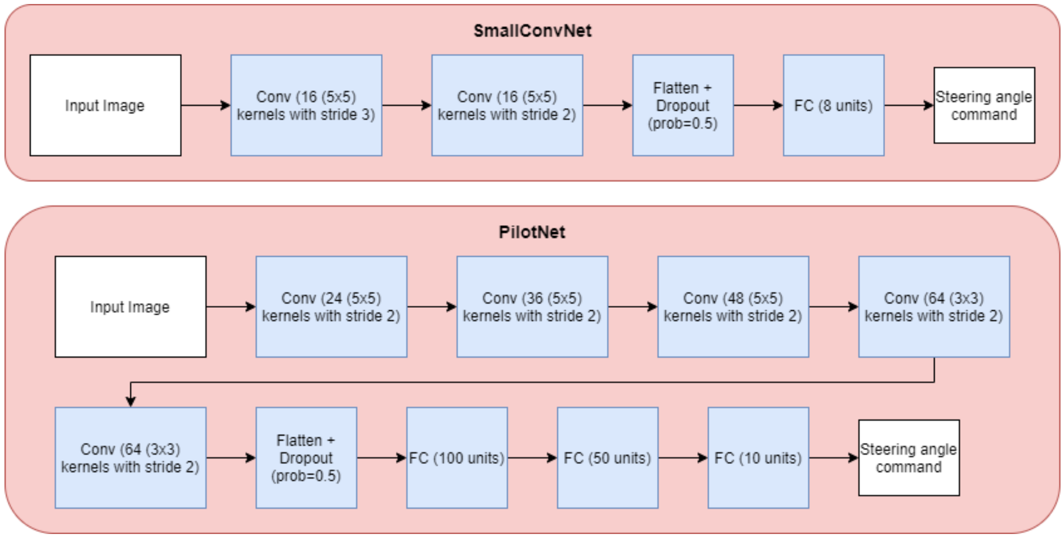

Figure 1: The two convolutional neural network architectures we use for purely feed-forward autonomous steering control.

- *Neural Network Interfacing*: The websocket server allows Python scripts to programmatically control the car while receiving images, which thus enables visual neural networks to control the car.
- *Telemetry*: While driving in autonomous mode, the simulator server also provides car telemetry data, which not only aids in autonomous control, but also allows for easy evaluation data collection.

3.3 Data Collection

Using the simulator, we collect two driving data sets in Training Mode by driving around the track for a total of five laps each around and on the track while recording front/lateral camera view data and telemetry data. For the first data set, we drive normally, as one would in a real driving scenario. This data is used to train and evaluate our neural networks' baseline performance in simulated self-driving tasks. In the second, we induce intentional, sub-optimal shakiness by frequently oscillating the steering angles (i.e. rapidly tapping the left and right arrow keys). This is done even on straightaways and turns, resulting in a snaking motion. Note that, even though this data is shaky, the car remains on the road for the entire data collection period and successfully completes the requisite five laps in this case as well.

3.4 Network Architectures and Smoothing Solutions

We train four different neural network architectures to imitate the human driver steering angles in the shaky data set.

- SmallConvNet: A shallow, low-parameter purely feed-forward convolutional neural network.
- PilotNet: A large convolutional neural network developed by NVIDIA.
- Standard RNN: A standard recurrent neural network with a densely connected recurrent cell, shallow convolutional header, and 8-unit state.
- LSTM: The same as the standard RNN, but with a long short-term memory recurrent cell.

Lastly, we also implement a averaging filter as an intuitive way of smoothing the policy commands. This is used in conjunction with the SmallConvNet.

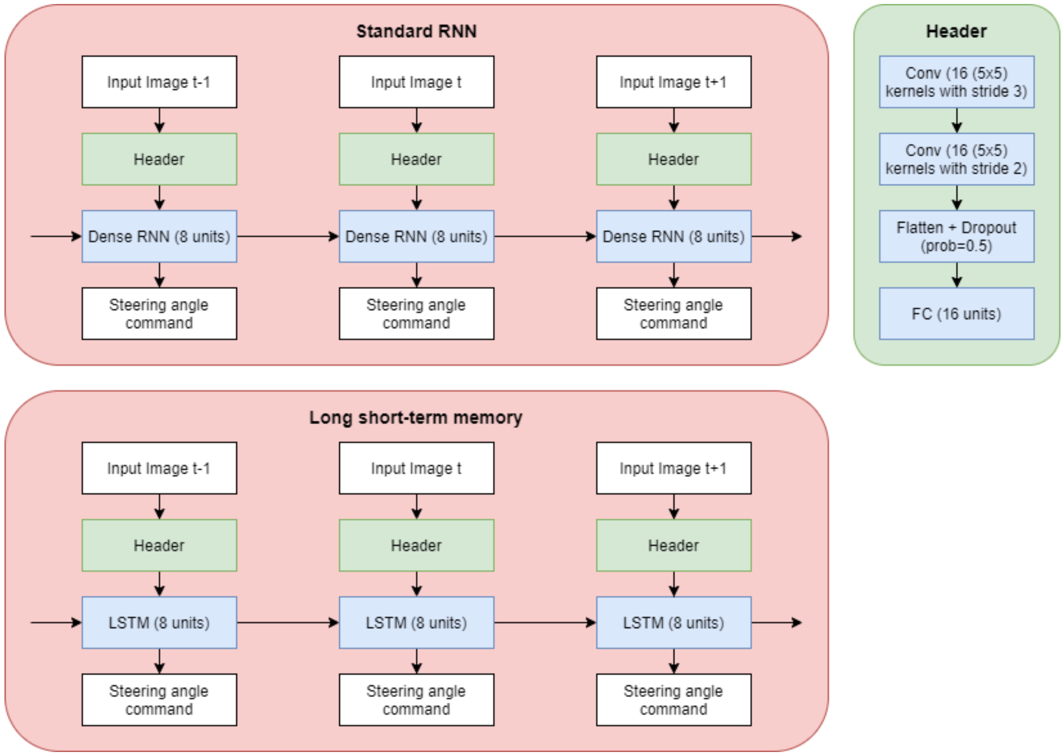

Figure 2: The two recurrent neural network architectures we use.

3.5 Software Implementation and Training

We implement the above neural networks with TensorFlow's Keras wrapper. We then train the networks using the Adam optimizer with mean square error as our loss function. We use a batch size of 64, an image sequence length of 16 for the recurrent networks, a learning rate of .0001, 20000 images per epoch, and 10 epochs total. We also make use of early stopping, deploying whichever training epoch checkpoint has the lowest loss.

During training for convolutional networks, we randomly perform the following data augmentations:

- Horizontally flipping the image and negating the ground truth angle so the network can learn to turn in both directions equally frequently
- Using a lateral camera's image and adding ± 0.2 radians to the ground truth angle to simulate the car recovering from almost driving off the road (which may not appear in the data set much) – Note: not used for recurrent networks
- Translate the image and add the scaled x-translation to the angle
- Add shadows to the image
- Scale the brightness of the image

3.6 Evaluation

To evaluate the networks' effectiveness, we allow each combination of model trained on each dataset drive the car around the simulated track in Autonomous Mode for one lap. During this time, we collect and save the commanded steering angles for analysis. We use 2 basic metrics to determine effectiveness. First is simply whether the car completed the track without falling off, thus showing clear indication that the method cannot learn a policy able to execute the most basic needs of a task.

Second, we examine the steering angle shakiness over the course of the lap. For this paper, we define shakiness as the largest change in steering angle in a certain lookahead time t_0. More formally, shakiness is defined as:

	LSTM	Basic RNN
Shaky Datset	5.14	4.94
Smooth Dataset	9.25	15.71

Table 1: Time in seconds that the recurrent models ran the car of the track.

$$\begin{aligned} s(t) &= min(k) \quad \text{s.t.} \\ k(\tau - t) &\geq y(\tau) - y(t) \quad \forall \tau \in [t, t + t_0] \\ -k(\tau - t) &\leq y(\tau) - y(t) \quad \forall \tau \in [t, t + t_0] \end{aligned}$$

where $y(t)$ is the steering angle at time t.

4 Results

For evaluation of our methods, we will first discuss the success of our methods. Then we will discuss the shakiness of the methods able to complete one lap around the track.

4.1 Success Rate

The only methods that failed were the RNN and LSTM models. However, all 4 of these models (i.e. each combination of RNN and LSTM trained on smooth and shaky data) failed. Table 1 describes how quickly each model failed in running the car off the track. Notice that for each architecture, the shaky dataset failed in significantly less time than the smooth dataset. This suggests that even when the car fails across the board, the suboptimal driving seems to confuse the model more than the smooth dataset.

4.2 Shakiness

Now we investigate the shakiness of each combination of architecture and dataset for those that successfully completed a lap around the track. Figure 3 shows a comparison of the shakiness of these varoius combinations. There are several patterns worthy of note. Unsurprisingly, across the board the shaky dataset induced more shakiness in the model policy. Therefore, as expected we can see the trends across architecutres more clearly in models trained on the shaky dataset than the smooth. Second, we can notice that the Small CNN with a filter produced the lowest shakiness by far across both datsets. Seeing as these methods both completed a lap successfully and are least shaky, we can suggest a simple moving average filter to be a very effective method of decreasing high frequency noise in policies without severely impacting policy performance. Lastly, we see the PilotNet experience less shakiness in the shaky dataset as compared to the baseline small CNN, but

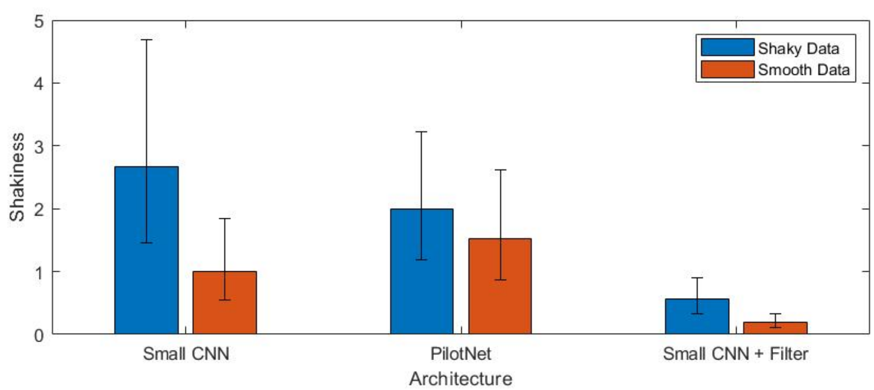

Figure 3: The sakiness of each combination of dataset and successful architecture.

more shakiness in the smooth dataset compared to the small CNN. More experimentation and insight is required to say for certain why this occurs. However, we believe this may result from the larger PilotNet having the complexity to abstract away the differences in dataset and converge to similar policies when compared to the smaller architectures.

5 Discussion

With the data above in mind, we can consider how to design models for physical systems, and what this project may overlook. Given Figure 1, we can tell that a simple moving average filter can provide a radical decrease in high-frequency noise in policy. However, it is worth remembering that the moving average is a tradeoff between smoothness and momentary precision. Our choice of task – a car driving around a track without many sharp turns – is relatively invariant to this type of precision. However, in different situations that may have instances where momentary sharp change is sometimes necessary, a moving average may not suffice. In the future, it may be interesting to investigate whether methods that attempt to learn from sub-optimal demonstrations such as D-REX [**?**] and SSRR [**?**] can also mitigate high-frequency noise. Lastly, in the future, we may also want to extend these methods to Reinforcement Learning methods, as shaky policies also show up in these environments. However, we would expect the moving average filter to work about the same as in the task described in this project. This is because the moving average filter is not a modification of the underlying model, but rather an effect placed at the end of the entire design and training process. Of course, we would need to tests these situations to be sure.

References

[1] Bojarski, S., Yeres, P., Choromanaska, A., Choromanaski, K., Firner, B., Jackel, L., & Muller, M. (2017) Explaining How a Deep Neural Network Trained with End-to-End Learning Steers a Car. Retrieved from https://arxiv.org/pdf/1704.07911.pdf

[2] Raffin, S. (2019) Learning to Drive Smoothly in Minutes. Retrieved from https://towardsdatascience.com/learning-to-drive-smoothly-in-minutes-450a7cdb35f4

[3] Shibuya, N. (2017) Introduction to Udacity Self-Driving Car Simulator. Retrieved from https://naokishibuya.medium.com/introduction-to-udacity-self-driving-car-simulator-4d78198d301d

[4] Brown, D., Goo, W., Nagarajan, P., & Niekum, S. (2019) Extrapolating Beyond Suboptimal Demonstrations via Inverse Reinforcement Learning from Observations. Retrieved from https://arxiv.org/pdf/1904.06387.pdf

[5] Brown, D., Goo, W., & Niekum, S. (2019) Better-than-Demonstrator Imitation Learning via Automatically-Ranked Demonstrations. Retrieved from https://arxiv.org/pdf/1907.03976.pdf

[6] Chen, L., Paleja, R., & Gombolay, M. (2020) Learning from Suboptimal Demonstration via Self-Supervised Reward Regression. Retrieved from https://arxiv.org/pdf/2010.11723.pdf

[7] Fu, J., Luo, K., & Levine, S. (2018) Learning Robust Rewards with Adversarial Inverse Reinforcement Learning. Retrieved from https://arxiv.org/pdf/1710.11248.pdf

6 Contributions

Alex Cuellar

- Debugged and modified neural network architectures
- Developed policy averaging filter
- Trained neural networks
- Tested and evaluated neural networks and filter
- Authored introduction, results, and discussion sections

William Chen

- Set up self-driving simulator and Python virtual environment
- Developed the neural network architectures in TensorFlow

- Developed and modified training and driving pipeline
- Collected smooth and shaky data
- Authored abstract, related works, and methodology sections