Robust Deep Reinforcement Learning for Autonomous Driving

Guillaume Chhor [*1] Shreyash Pandey [*2] Vivekkumar Patel [*3]

Abstract

Self-driving cars are set to transform the transportation industry in a few years. While a lot of end-to-end supervised deep learning approaches have achieved success in making cars autonomous, the compounding errors that exist in the environment compel us to employ a Reinforcement Learning approach to solving this problem. We present here our results in using deep reinforcement learning methods to tackle the challenge of autonomous driving. Considering the complexity of the task we describe how we are using the TORCS environment for simulation, and a Deep Deterministic Policy Gradient approach to train our autonomous race car. We introduce ways and methods for training the algorithm so that it is robust to facing noisy sensor inputs, a common issue when dealing with complex hardware systems, without significant loss of performance as compared to an ideal noiseless system.

1. Introduction

Self-driving cars is a topic of great interest in the AI field. Since the 1920s, experiments have been conducted and numerous breakthroughs have been achieved. Fully autonomous cars today use sophisticated techniques such as lasers, radar and various others. We wish to see if Deep Reinforcement Learning Techniques could be used to solve this task effectively and efficiently, with a particular focus on how they behave in presence of noisy input signals, and how to cope with any problems that may arise.

Reinforcement Learning (RL) techniques have been successful in teaching machines to interact with the environment and learning from the experience. It has been successfully applied to games like Atari and Go by Google DeepMind. In this work, we apply these techniques to the problem of autonomous driving, the motivation being that this problem too has high dimensional input, which RL agents have been demonstrated to be good at. We also test their robustness to noisy sensors, and suggest ways of making it more robust.

[*]Equal contribution [1]ICME, Stanford University [2]EE, Stanford University [3]CS, Stanford University. Correspondence to: Guillaume Chhor <gchhor@stanford.edu>, Shreyash Pandey <shreyash@stanford.edu>, Vivekkumar Patel <vivek14@stanford.edu>.

2. Related Work

Most of the research on self-driving cars was inspired by the pioneering work of (Pomerleau, 1989) who in 1989 built the Autonomous Land Vehicle in a Neural Network (ALVINN) system. It demonstrated that an end-to-end trained neural network can indeed steer a car on public roads. The DAVE-2 (Bojarski, 2016) system from NVIDIA followed in the footsteps of ALVINN by training a powerful Convolutional Neural Network that maps raw pixels from a single front-facing camera directly to steering commands. Most of such ”perception” based approaches are classified as ”behavior reflex approaches” that directly map an input image to a driving action by a regressor. Other methods such as Deep-Driving (Chenyi Chen, 2015) propose to map an input image to a small number of key perception indicators that directly relate to the affordance of a road/traffic state for driving.

The underlying principle of supervised learning in all these approaches is extremely prone to compounding errors, wherein one bad action at a particular timestep can lead to a state that is too hard to recover from and hence affects all the future decisions as well. Reinforcement Learning is the perfect paradigm to model such environments, and a lot of research has been conducted to employ Deep Q Learning (DQN), or Policy Search approaches that maximize a reward, most generally achieved at the end of the race. These approaches lead to a policy that maximizes future rewards, and hence is less prone to compounding errors.

We also reviewed previous years' CS231N project report (April Yu, 2016), which was a great starting point as it surveyed a few basic approaches such as DQN and it's variants. Their report establishes that DQN can be effective in discrete control problems. A lot of work has been done to solve autonomous driving using continuous action spaces. Deep Deterministic Policy Gradient (DDPG) (Lillicrap, 2016) and Trust Region Policy Optimization (TRPO) (Schulman J., 2015) are some of the typical ones. (Zhang & Cho, 2016) looks at solving this problem using the imitation learning approach, where they propose learning by iteratively collecting training examples from both reference and trained policies. This makes sure that there is minimal unexpected behaviour due to the mismatch between the states reachable by the reference policy and trained policy functions. One drawback of imitation learning though, is that it requires the help of the reference policy at all times.

In this work, we present a DDPG approach to solving this problem, where we use standard reinforcement learning methods such as prioritized experience replay, target networks and Ornstein Uhlenbeck exploration to train an agent to predict steering, brake and acceleration values from continuous spaces. The problem with training Reinforcement Learning agents on simulators is that they are not directly transferable to the real world because no sensor is ideal, and there is always some form of noise that our model has to deal with. To this end, we conduct a thorough robustness analysis of the model by testing it at different levels of noise, and also suggest and implement ways of making it more robust.

3. Approach

3.1. TORCS Simulator

Due to the nature of this problem, it is impossible to conduct the experiments and training with a real car. Hence, we use TORCS as our simulator due to it's OpenAI gym like interface.

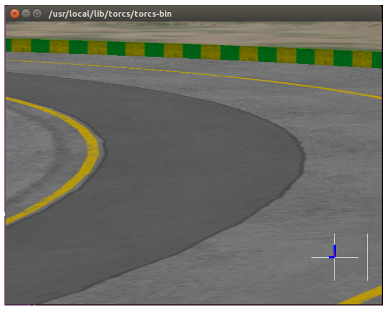

Figure 1. Visual display in TORCS

TORCS provides 18 different types of sensor input but generally the following inputs are considered most useful:

- Angle between the car direction and the direction of the track axis.
- Speed of the car along the longitudinal axis of the car (good velocity).
- Speed of the car along the transverse axis of the car
- Speed of the car along the Z-axis of the car
- Vector of 4 sensors representing the rotation speed of wheels
- Number of rotation per minute of the car engine
- Distance between the track edge and the car within a range of 200 meters
- Distance between the car and the track axis

We used TORCS because of abundance of RL experiments available on GitHub upon which we could build our own. We encountered several issues along the process, it was notably only available in Linux and no display was available on Azure, making the task of training the algorithm from the visual input on GPU impossible. That is the main reason why we decided to conduct our experiments with the sensors inputs.

3.2. Deep Deterministic Policy Gradient

To deal with continuous action spaces, DeepMind came up with policy-gradient actor-critic algorithm called Deep Deterministic Policy Gradients (DDPG) which is a off-policy and a model-free RL algorithm. It employs a stochastic behavior policy for exploration, but learns a deterministic target policy, which is much easier to estimate. Policy gradient algorithms are akin to a form of policy iteration: they evaluate the policy, and then use the policy gradient to improve the value. DDPG is an instance of actor-critic paradigm in that it employs two neural networks, one for the actor that acts and one for the critic, which evaluates actor's decisions. More specifically, the actor network computes action predictions for the current state and the critic network generates a temporal-difference (TD) error signal based on the reward at each time step. The actor network takes the current state as the input, and outputs a real value representing an action sampled from a continuous action space. The critic takes as input both the state, and the action predicted by the actor, and outputs the estimated Q-value of the current state and of the action chosen by the actor. It is important to note DDPG is off-policy and learns a deterministic target policy, and this allows for the use of the celebrated Deterministic Policy Gradient theorem by (Silver et al., 2014) which provides the update rule for the weights of the actor network:

$$\nabla_{\theta^\mu}\mu \approx \mathbb{E}[\nabla_a Q(s,a|\theta^Q)|_{s=s_t,a=\mu(s_t)}\nabla_{\theta^\mu}\mu(s|\theta^\mu)|_{s=s_t}]$$

And the critic network is updated from the gradients obtained from the TD error signal:

$$y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$$

$$L_{critic} = \frac{1}{N}\Sigma_i(y_i - Q(s_i, a_i|\theta^Q))^2$$

Here, L_{critic} is the loss function for the critic network.

The agent receives the sensor input in the form of array which is fed into a policy neural network, and the network outputs three real numbers (value of the steering, acceleration and brake). The policy network is trained via the Deep Deterministic Policy Gradient, to maximize the future expected reward.

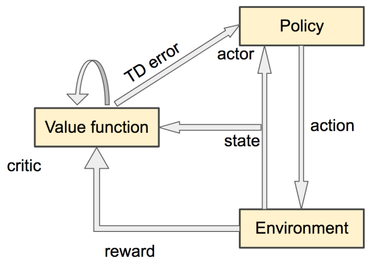

Figure 2. Interactions in an Actor Critic Algorithm

3.3. Robustness of DDPG to noisy sensors

We consider sensor inputs as our state for this problem. Even in real scenarios, some of the different sensor inputs mentioned in the previous section, are used as an input to the agent. One of the very common problems with electrical sensors and circuits is of Additive White Gaussian Noise. Hence, in this project, we wanted to see the robustness of the DDPG algorithm to this noise and decided to check for 4 different levels of noise, specifically for standard deviation in [0.2,0.4,0.6 and 0.8] (note that white gaussian noise has zero mean). This choice was made because all our sensor inputs are normalized to lie in the range [0,1], and hence, these values of standard deviations would amount to significant level of noise power in the input signal.

While experimenting with a noise of standard deviation of 0.2, we observed that the agent failed to learn a good policy. Hence, we dropped the idea of probing with higher noises and instead, focused on probing more in the [0-0.2] range. We train the agent on different noise levels and test them on a different track with same noise levels. The results of these experiments have been summarized in the next section.

3.4. Additive White Gaussian Noise

White Gaussian noise is a common way to model noises. The following explains the terminology:

- White: A white noise means that the power spectrum is constant (i.e power is same at all frequencies).
- Gaussian: Effects of different kinds of noises gets added (thermal noise, noise due to discretization, sampling errors etc.). Using the Central Limit Theorem, we can approximate the total noise using a Gaussian distribution. Hence, specifying the mean and variance will characterize the whole distribution.

As the noise is white and follows gaussian distribution, the time samples are completely uncorrelated, i.e noise at time $t + 1$ is completely independent of noise at all the previous time-steps $0, 1, ...t$.

4. Experiments, Results and Observation

4.1. Implementation Details

4.1.1. Prioritized Experience Replay

We use prioritized experience replay for two reasons. It helps break temporal correlations on the training samples and make them more i.i.d like samples. This also helps in using previous samples numerous times by storing them, instead of training for more number of episodes. Another advantage is that as we have a huge batch of observations available to sample from and so we can sample a batch of observations to make use of hardware efficient minibatches for training. The difference between normal experience replay and prioritized experience replay is that in prioritized experience replay, the observations are sampled according to a non-uniform distribution, generally using the t-d error for a sample as the probability mass for that sample.

4.1.2. Target networks

We use two target networks, one for actor and one for critic. Target networks help in stabilizing the training as they prevent positive feedback loops in the network. The target networks are exactly the same as the online networks, but they are updated less frequently (or at a slower rate) than their online counterparts. In the equations in the previous section, μ and θ denote the online network whereas μ' and θ' denote the target networks.

4.1.3. Exploration Exploitation

It should be noted that vanilla ϵ-Greedy policy does not work well in tasks such as these because we have a 3-dimensional action (steering,acceleration,brake) and randomly sampling the action from a uniform random distribution will generate some nonsensical combinations, for instance the value of the brake could be greater than the value of acceleration which would make the car stop.

For exploration during training, we instead use a Ornstein Uhlenbeck process whose level depends notably on the definition of an ϵ-Greedy parameter and that is defined as follows:

$$OU_{\theta,\mu,\gamma}(action) = \theta(\mu - action) + \gamma N(0,1)$$

$$action := action + \epsilon\text{-}Greedy \times OU_{\theta,\mu,\gamma}(action)$$

By keeping μ higher for acceleration as compared to that of brake, we can make sure that the car never stops in the exploration process. This form of exploration also prevents the model from begin stuck in a local minimum and enables ”smoother” exploration. We make this process decay over time by decreasing the ϵ-Greedy parameter, favoring exploitation after a certain time.

4.1.4. Network Architecture

The actor network consists of two hidden layers and maps sensor inputs to steering, acceleration and brake values, whereas the critic network concatenates the state encoding (hidden layer representation for sensor inputs) with the actor’s prediction and then employs one hidden layer to calculate the Q value.

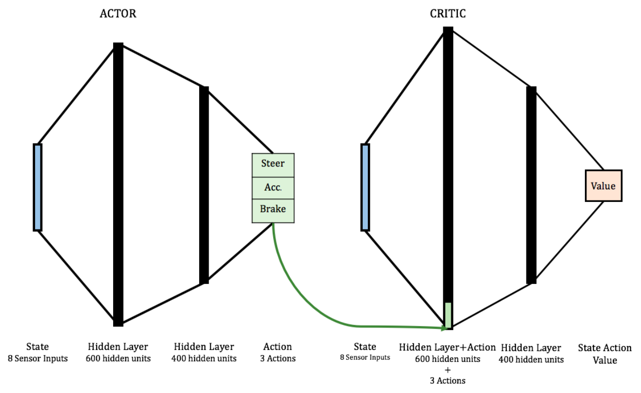

Figure 3. Network architecture

4.1.5. Training Details

For training we used Adam optimizer with a higher learning rate for critic than actor, in order to make sure updates to actor network are stable.
The reward function is defined by the following formula:

$$R_t = V\cos\theta - V\sin\theta - V|trackPos|$$

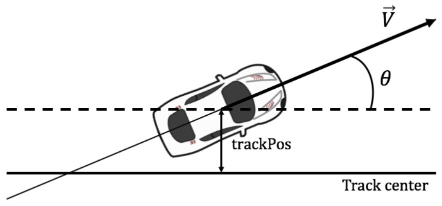

Figure 4. Reward design

The intuition behind is that it maximizes velocity along the track, minimizes velocity along the transverse axis and encourages the car to remain in the center of track if its speed is high. During a turn, when the car moves away from the track axis, this reward function will force the car to slow down. Note that the $trackPos$ variable is normalized so that all the terms lie on the same scale.
A possible improvement to this current reward would be to take into account and penalize any damages made to the car, forcing the agent to drive safely.

4.1.6. Experiments with Noisy Sensors

Finally, we added noises of different levels to the sensors input to reproduce this common phenomenon in actual sensors, during training and testing and experiment with new architectures that should increase the robustness of the agent to the noise.

- We check the robustness of DDPG to different levels of noise by checking the performance of our agent on repeated trials with values of noise std-dev ranging between 0 and 0.2 (4% noise power).
- We compare the performance of agents trained on noiseless environments with that of an agent trained with noise, on both noisy and noiseless environments.
- We compare the training time of agents trained with different levels of noise to gauge the difficulty of learning from highly noisy sensor inputs.
- We experiment with different neural network architectures by stacking more layers in the actor and critic networks.
- We provide multiple noisy samples to actor and critic networks by concatenating noisy estimates of sensor inputs a fixed number of times before feeding it to the networks. This is akin to having a larger number of sensors, or to having a sensor with a higher sampling rate that generates more estimates per unit of time. The idea is to see if the model is able to figure out that averaging the multiple samples leads to reduced variance of noise.

4.2. Results and Observations

4.2.1. Robustness Analysis

We train and test for different standard deviations of noise. The training and testing is on different tracks.

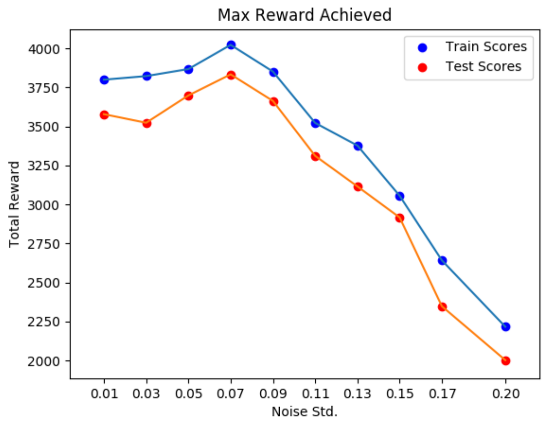

Figure 5. Training and testing with different levels of noise

In Figure 5, X-axis is the level of noise σ, with the inputs normalized in order to have unit magnitude. The training and the testing was done on different tracks to demonstrate the agent's ability to generalize its learning.

We can notice that a small noise at around $\sigma \approx 0.07$ helps the agent to perform better, but too much noise makes the algorithm break down at $\sigma \approx 0.20$. This can be explained due to the following facts:

- As we add noise directly to the sensor input, this essentially changes the state that the agent sees. Hence, adding a small noise helps the agent to explore the state space more.
- For input features such as positions of obstacles, road edges and other vehicles, adding a small noise models a small uncertainty in their position. Due to this, the agent would try to learn a safer policy and will avoid collisions by keeping safe margins between itself and other obstacles or road edges. This could be a crucial aspect in real life scenarios as well.
- As the noise standard deviation is increased, the noise becomes too large for the agent to make sense of any input that it gets. It is no longer sure about its position and that of other objects as well. Hence, the algorithm will always break down.

4.2.2. Increased Robustness

Train — Test	**No Noise**	**Noise** ($\sigma = 0.1$)
No Noise	3579	2340
Noise ($\sigma = 0.1$)	3137	**3312**

Table 1. Performance of Algorithms with and without Noise

Table 1 compares the max reward of the agent trained on noiseless environments with that of an agent trained with noise, on both noisy and noiseless environments at test time. The results show that the agent trained with noise is, as expected, more robust at test time in presence of noise than an agent trained without noise. But moreover it performs almost as well as the agent trained without noise in an environment without noise. We can conclude that training with a small noise makes the agent more robust and gives better performance.

4.2.3. Training Time Comparison

Figure 6 shows that as the noise power increases, it takes longer for the algorithm to converge. The blue curve indicates a noise with $\sigma = 0.05$ and the green one indicates a noise with $\sigma = 0.13$.

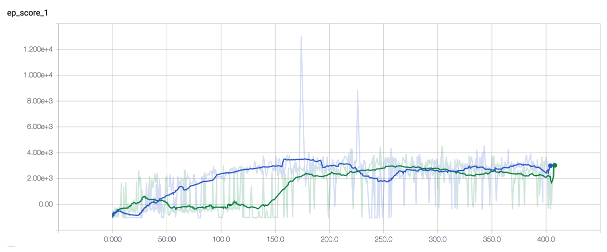

Figure 6. Training Curves with two different noises

4.2.4. Architectural Changes

Our experiments with trying to improve the complexity of the architecture by adding extra layers to the actor and critic networks did not bear fruit as the algorithm either failed to converge, or stagnated at a negative reward after a few iterations. This could be due to a variety of reasons. With a more complex parameter space, our policy gradient approach is more susceptible to local optima, and hence is likely to stagnate early. However, the possibility of "cascading failure" is more likely and the presence of noise is perhaps leading to gradients that are too high sometimes leading to failure to converge, or too low at other times leading to stagnation.

4.2.5. Multiple Samples Architecture

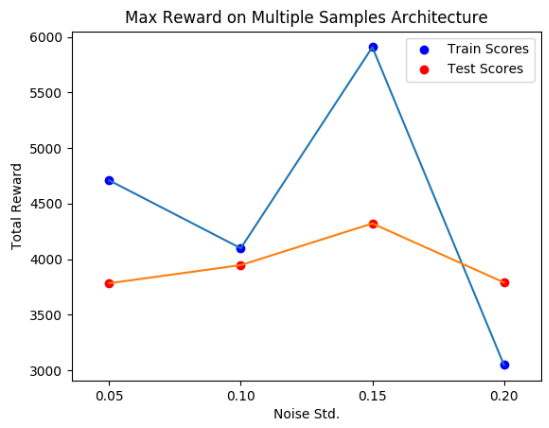

Figure 7. Training Curves with multiple samples of noisy input

Figure 7 shows the results of our multiple samples architecture. In this, we make T (10 in our case) copies of the sensor values, independently add noise to them, and concatenate the inputs to T-fold increase both the input size and the weight matrix for the first hidden layer in the actor and critic networks. The idea was to mimic having T noisy sensors instead of one, or having a sensor with a T-fold sampling rate. Not surprisingly, our model was able to make use of the wealth of information it was provided with, with max scores much higher than the single sample case, for both training and testing. A quick look at the weights of the first hidden layer of actor and critic shows comparable mean and std-dev of columns of the weight matrix that correspond to the same sensor type. This shows that our model has learnt to average the sensor inputs to reduce variance of the noise. One more observation is that our model now achieves the best performance at a higher noise std-dev than before (0.15 as compared to 0.07). It is to be noted that a T-fold decrease in variance due to averaging the sensor inputs makes both the cases almost identical. Hence, this architecture serves as a good solution for the higher noise problem. This is also consistent with our hypothesis that a little noise is good for exploration.

5. Conclusion

From what precedes, we saw that adding a little noise during training can improve the performance of the agent as well as make it more robust when dealing with noisy sensor inputs. This result can be interpreted as the inherent noise of the sensors being an additional form of exploration. We also noticed that addition of noise during training makes the learning harder, and our model takes a long time to train. Architectural changes in the form of more layers in the actor and critic network failed to converge to good results, which we attribute to ”cascading errors” due to noise. Finally, we come up with a multiple samples architecture that takes as input a concatenated vector of multiple noisy sensor estimates and learns to average them to reduce their variance. This makes our model significantly robust to additive white gaussian noise, making it transferable to the real world.

6. Future Work

To go further, an interesting thing to do would be to implement other models such as TRPO or DQN and test their robustness when facing a noisy environment. This will facilitate fair comparison between the different models in terms of their performance, as well as establish a benchmark of their robustness. Also, if a little noise helps in improving the performance of these other models, then it will be possible to generalize the benefits of a small amount of noise.

Finally, as an alternative to using noisy sensors as input, the agent could rely only on visual inputs that are less prone to this kind of noise. Unfortunately, with the current version of TORCS, it is difficult to work with visual input on a system like Azure which does not have display.

One more direction could be a theoretical analysis to get bounds on the performance estimates based on the level of noise in the input. (Jun Morimoto, 2001) talks about an Actor-disturber-critic model wherein they model noise in the input, and construct gradient updates to the function approximators that incorporate the level of noise present in the input. It would be interesting to see how the presence of noise in the gradient updates affects the reward that we are maximizing. We are interested in obtaining bounds on the max rewards attainable in that case.

Acknowledgements

The authors would like to thank James Harrison for his precious help, advice and ideas for this project. We would also like to thank only4j, as we have done our work by building upon his existing github repository.

References

April Yu, Raphael Palefsky-Smith, Rishi Bedi. Deep reinforcement learning for simulated autonomous vehicle control, 2016.

Bojarski, Mariusz. End to end learning for self-driving cars.

2016.

Chenyi Chen, Ari Seff, Alain Kornhauser Jianxiong Xiao. Deepdriving: Learning affordance for direct perception in autonomous driving. 2015.

Emami, Patrick. *Deep Deterministic Policy Gradients in TensorFlow*. URL `http://pemami4911.github.io/blog/2016/08/21/ddpg-rl.html`.

Jun Morimoto, Kenji Doya. Robust reinforcement learning. *NIPS*, 2001.

Kim, Daejoong. *Deep Reinforcement Learning*. URL `https://github.com/futurecrew/DeepRL`.

Lau, Ben. *Using Keras and Deep Deterministic Policy Gradient to play TORCS*. URL `https://yanpanlau.github.io/`.

Lillicrap, Timothy P., et al. Continuous control with deep reinforcement learning, 2016.

Pomerleau, Dean A. Alvinn, an autonomous land vehicle in a neural network, 1989.

Schulman J., Levine S., Abbeel P. Jordan M. Moritz P. Deep reinforcement learning for simulated autonomous vehicle control, 2015.

Silver, David, Lever, Guy, Heess, Nicolas, Degris, Thomas, Wierstra, Daan, and Riedmiller, Martin. Deterministic policy gradient algorithms, 2014. URL `http://dl.acm.org/citation.cfm?id=3044805.3044850`.

Zhang, Jiakai and Cho, Kyunghyun. Query-efficient imitation learning for end-to-end autonomous driving, 2016.