Driving in Duckietown using Reinforcement Learning

[redacted]

Abstract

This paper explores the application of reinforcement learning to the popular problem of autonomous driving. We use popular actor critic reinforcement learning methods that teach an agent how to learn to navigate and avoid collisions from images. Using end-to-end learning we leverage the power of convolutional neural networks to process the state and then act as both the policy and the value function. At each step of development, we noted the possible improvements by looking at how the rewards changed during training and how the policy performed after training. We did gradual improvements, leading to our best result with the TD3 algorithm. As our baseline, we use the DDPG algorithm which resulted in an average reward as 923 and subpar driving. We then improved on our baseline, which gave us reward as 1312 and a policy that learned that driving in circles the entire time would result in better rewards because it would not hit anything and always be in the center of a lane. We also came up with a variation involving training a convolutional encoder to encode the state to a vector of size 480 and using the embeddings to train policy and Q-value networks using DDPG to obtain a final average reward of 1273. Then, we implemented the TD3 algorithm, a popular improvement on DDPG, and reshaped the reward function which gave us a lower reward (due to changing the how the rewards are calculated) but gave us much better driving. The policy learned with TD3 was able to drive in a straight line and gave an average reward of 309.

1 Introduction

We tackle the open problem of autonomous driving with the application of reinforcement learning (RL). This problem is being researched upon extensively and end-to-end training in simulated environment to then migrate onto a real world scenario is the most popular one as described by zhang et al. in [1]. Training policies using in an off-policy setting is also a ver popular approach and its capabilities have been showcased by Vitelli et al. in [2]. We take these papers as inspiration and work on our project where we train policy and Q-networks end to end in an off-policy setting.

Since the experiments cannot be run in real world due to the number of trials it takes for the agent to start performing desired behavior and the cost associated with it, there are a lot of simulators which can simulate real world scenarios. CARLA [3] and Deepdrive [4] are two such simulators which are state of the art and provide almost real-world renderings and sensor data. But we use Duckietown, an open sourced autonomous driving simulator, as our environment [5] due to our computation and time constraints. The simulator has an agent, a Duck car, that has to drive around Duckietown, navigating roads with intersections, obstacles, pedestrians, and other Duck cars. In reinforcement

learning, the dataset is generated by interactions of an agent with its environment. As the Duck car moves around, data in the form of images are collected to train the reinforcement learning algorithm.

Deep Deterministic Policy Gradient (DDPG) [6] was considered to be the baseline algorithm. DDPG is an actor-critic RL method, where an approximator of the value function of state and action pairs (critic) and an approximator of the policy function are both modeled as neural networks and learned with off-policy data from a replay buffer. In this, the actor and critic each have their own feature extractor which is trained separately for each to generate a representation of an image (the state). We also extended DDPG and implement Twin Delayed DDPG (TD3) [7]. While DDPG is a popular and successful algorithm, it is also hard to train and very sensitive to hyperparameters. TD3 improves on DDPG by learning two Q-functions instead of one, delaying updates to the policy function, and adding noise to actions.

We also experimented with a few variations on the simulator. We edited the original reward function given by the simulator to motivate driving forward. We tried training on wheel torques (modeling the Duckiebot as a differential drive model) and velocity with steering angle (modeling it as a steering angle velocity control model) as an action space. We also tried to extract features from an image with an autoencoder, then train the agent with states from the encoder, as a means to reduce the training time.

In following section, we will discuss the motivation behind the work we have done. In section 3, we discuss the methods we used. In section 4 and 5, we will go over the results and discuss them.

2 Motivation

The self-driving car is a vehicle that can process its surroundings and navigate properly without any human input. While there are vehicles on the road currently with technology that has some resemblance of autonomy, it has not been perfected yet and still require human interaction (such as the requirement of hands on the steering wheel). Because this is still an unsolved problem, teeming with public interest, we want to understand and attempt to solve a simpler version of it with reinforcement learning. The main problems with existing systems are the volume of multi-modal data available, speed of detection since this ties in directly with safety of the system, and complexity of detection based on environmental anomalies.

Reinforcement learning is built upon the the notion from behavioral psychology, that states that a person's future behavior can be influenced by a consequence or reward. Without even having to delve into the field of psychology, this notion should intuitively make sense. We as human beings learn in many different ways and one way is through trial and error where we learn from our mistakes or our successes. It should follow that we can teach a car to drive itself in a similar fashion. While we can not have an algorithm design a control policy by driving a car in the streets of La Jolla, we can have a simulated duck car in a simulated street occupied with ducks. In this paper, we will explore the ability of this simulated car to navigate this duck-inhabited city by only using a single camera. Using a simpler animal such as a duck instead of a vehicle, helps us trivialize movement related stimuli to a simple steering and a velocity value at every timestamp. This helps us explore actual complexities in these problems, while ensuring implementation is optimized for sporadic decision making based on input stimuli. While current semi-autonomous vehicles have been made with a multitude of sensors, from inertial measurement units to lidar to GPS, we will focus on the camera as our only sensor. This way, we can attempt to learn a driving policy end to end, from pixel to action, without engineering valuable states that are deemed useful for the algorithm and harness the ability to learn useful features in service of the task with convolutional neural networks.

3 Methods

3.1 Preliminaries

We will first go over useful terminology and some background information on reinforcement learning that was useful in building our approach and potentially to the reader.

3.1.1 The RL Problem

The key terms used to describe a RL problem are: agent, environment, state s, action$a$, reward$r$, policyπ, value$v = Q(s, \pi(a))$. Let us define each of them for our particular problem:

1. **Agent**: Duckiebot, which is modeled as a differential drive robot.
2. **Environment**: Duckie-town. This is a virtual town containing roads with intersections, obstacles, pedestrians, and other Duckiebots. The virtual environment is also broken into tiles that dictate the location of the agent in the environment, as shown in the figure 2. The environment dictates that a run is over if the Duckiebot hits an obstacle, leaves the lane, or reaches a maximum of 500 steps.
3. **State -** s: 2D images of the environment captured at 640×480 resolution by the agent, at a rate of 30 fps.
4. **Action -** a: The environment contains two action settings where all the values are scaled with respect to their maximum and minimum values. i.e. -1 corresponds to the minimum allowed value and 1 corresponds to the maximum allowed value for each action variable. The two setting available are as follows:
   Setting1: A tuple of two numbers (τ_1, τ_2), each number describing the torque to be applied on the left and right wheel respectively. Both $\tau_1, \tau_2 \in [0, 1]$. Setting2: A tuple of two numbers (v, θ), where v represents the velocity of the agent and θ represents the steering angle. The bounds for the same are defined as $v \in [0, 1]$ where and $\theta \in [-1, 1]$.
5. **Reward -** r: The default reward function tries to encourage the agent to drive forward along the right lane in each tile. This reward is a dense reward, shaped to incentivize driving on the road and avoid collision with reasonable speed. The reward at time step t given by r_t is dependent on velocity(v_t) in the direction of the road at that point (d_t), distance from centre of the lane(l_t) and vicinity to obstacles (C_t) at time t as shown in the equation below:
   $$r_t = \alpha \vec{v_t}.\vec{d_t} - \beta l_t - \gamma C_t \tag{1}$$
   where, $\{\alpha, \ \beta, \ \gamma\} > 0$ are constants and the relative value between them gives how important each term is, and hence controls the behavior of the agent.
6. **Policy -** $\pi(s)$: A neural network which takes in the state s as an input and returns the action a. The optimal action to be taken at the given state is learn by training the network.
7. **Value function -** $Q^{\pi}(s, a)$: A neural network which takes in the state s and action a, and returns the expected long term reward for taking action a at s and following the policy π there after till the end of the episode. The estimate of this value is learn during training.

3.1.2 Policy-Gradient Method

Traditional RL methods involve computing the value/Q-function approximations at each state and determining the optimal action based on these values. But in a policy-gradient method, the policy is computed directly by computing the gradients of the expected reward of a policy, and updating the policy in the direction of increasing long term rewards. As seen in class, the problem with this vanilla-PG is that getting one reward signal at the end of a long episode makes it difficult to figure out exactly which was the good action. This is not very useful especially for RL problems with a continuous action space. There are extensions to this such as REINFORCE which makes the algorithm more useful.

3.1.3 Actor Critic Algorithms

In policy gradient algorithms, the policy function is represented independently from the value function. It has been proven that a vanilla policy gradient algorithm leads to high variance results and hence a sub-optimal policy. Hence a baseline critic is necessary to reduce the variance and to produce better, stable results. The policy function is referred to as the actor, and the value function as the critic. The actor produces an action given the current state of the environment, and given this state and the resultant reward, the critic produces an error signal. Essentially, the critic is estimating the Q(s, a) function based on the output of the actor. The output from the critic is used to drive the learning of the actor and critic. As these actor and critic functions are non-linear and complex

Figure 1: Simulated Duckietown Environment

for environments with continuous state-space and action-space, they are represented using neural networks. A general actor critic architecture is as shown in the figure below.

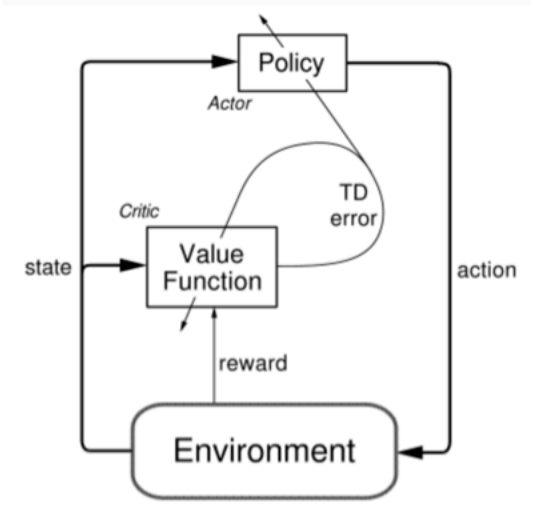


Figure 2: General Actor Critic Architecture

3.1.4 Off-Policy vs. On-Policy

For training the policy and Q-value functions we need to generate episodes $\rho = s_0, a_0, r_0, s_1, a_1, \ldots, r_{\tau-1}, s_\tau$ which start from the starting state s_0 and end at some terminal state s_τ. On-policy algorithms uses the policy π that is being learnt to sample data during training. Where as, off-policy algorithms use a separate behavior policy μ that is not dependent on the policy being improvedπ. The behavior policy μ can now operate by sampling all actions, while the estimation policy π can be deterministic or greedy. 'Q-learning' is an off-policy algorithm. Q-learning algorithm states that the Q-value corresponding to state s_t and action a_t is updated using the reward r_t and the Q-value of the next state corresponding to the action that is determined by the policy. i.e. $Q(s_{t+1}, \pi(s_{t+1}))$.

3.2 Baseline: DDPG

Deep Deterministic Policy Gradient(DDPG) is an off-policy deterministic policy-gradient method. DDPG is an actor-critic algorithm as it primarily uses two neural networks, one for the actor i.e. policy and one for the critic i.e. the Q-value function. Together, these two networks compute action for the current state and generate an error signal based on the reward and Q-value at each time step.

But then, having just the above two networks with Deterministic Policy Gradient results in an algorithm that does not converge easily. This is because, training the functions with a lot of correlated simulated trajectories leads to a lot of variance in the approximation of the critic. The error signal is very good at taking the variance produced by the bad predictions into account. So, using a replay buffer to store the experiences of the agent during training, and then randomly sampling from the experiences to use for learning really helps. As this helps in breaking up the time based correlations inside a training episode. This is also known as experience replay.

The deterministic policy gradient theorem provides the update rule for the weights of the actor network and the critic network is updated from the gradients obtained from the error signal. Updating the weights for both the networks based on the gradients obtained from the error signal that was computed based on both the replay buffer and the output of the actor and critic networks leads the learning algorithm to diverge. So, DDPG proposes to use a set of target networks to generate the targets for the error computation, which regularizes the learning algorithm and increases stability.

The equations for the TD's target and the loss function for the critic network are given below.

$$y_i = r_i + \gamma Q' \left(s_{i+1}, \pi' \left(s_{i+1} | \theta^{\pi'} \right) | \theta^{Q'} \right) \tag{2}$$

$$L = \frac{1}{N} \sum_i \left(y_i - Q \left(s_i, a_i | \theta^Q \right)^2 \right) \tag{3}$$

In the equations, a mini batch of size N is sampled from the buffer. with the index i refering to the i-th sample. The target y_i is computed by summing up the immediate reward, and the outputs of the target actor and critic networks characterised by the weights represented in the equation. The critic loss is then computed with the output of the critic network for the i-th sample.

The weights of the critic network is updated with the gradients obtained from the loss function as in the equation above, while the actor network is updated with the Deterministic Policy Gradient as given in the equation below:

$$\nabla_{\theta^\pi} J \approx \frac{1}{N} \sum_i \nabla_a Q \left(s, a | \theta^Q \right) \Big|_{s=s_i, a=\pi(s_i)} \nabla_{\theta^\pi} \pi \left(s | \theta^\pi \right) \Big|_{s_i} \tag{4}$$

Finally, all we need is the gradient of the output of the critic network with respect to the actions, multiplied with the gradient of the output of the actor network with respect to its parameters.

Also, the target network parameters updated by weighing the parameters of the target networks and that of the Q-value and policy networks using the equation:

$$\begin{aligned} \theta^{Q'} &\leftarrow \tau \theta^Q + (1 - \tau) \theta^{Q'}, \\ \theta^{\pi'} &\leftarrow \tau \theta^\pi + (1 - \tau) \theta^{\pi'}, \\ \tau &\ll 1 \end{aligned} \tag{5}$$

The architecture of the actor and critics used for baseline are as given below in fig 3. The CNN architecture is given in Table 1, the FCN of the actor contains 2 layers 1 with the output size of the CNN i.e. 4032 and the next layer of size 512. The critic on the other hand has 3 layers, the first layer with size 4032, second layer with a size of 256 and the third layer of size 128.

Layer	In	Out	Kernel	Padding	Stride
CNN(4-layers):					
Conv, BatchNorm	3	32	8x8	0	2
Conv, BatchNorm	32	32	4x4	0	2
Conv, BatchNorm	32	32	4x4	0	2
Conv, BatchNorm	32	32	4x4	0	1

Table 1: Convolutional Layers for all the actor and critic networks

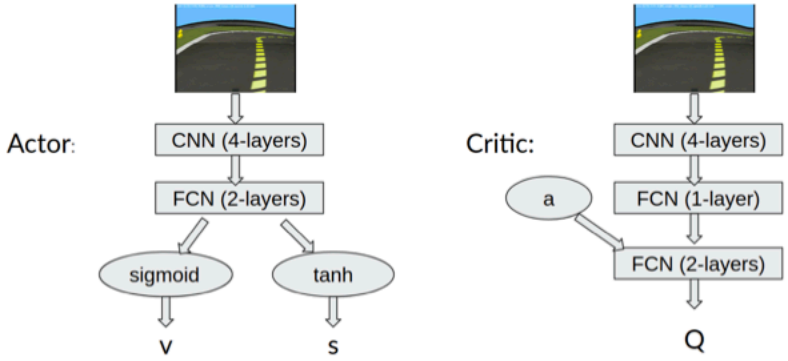


Figure 3: Baseline architecture

3.3 Change in Architecture

The architecture is changed and experimented with, in order to improve the results.

3.3.1 Changes to Actor and Critic network

In order to improve the results and make them converge faster to an optimal value, the following changes are made to the architecture:

1. The input a_t parameters are changed from velocity and steering angle to wheel torques (τ_1, τ_2) to avoid conversion from velocity and steering angle to torque which happens internally in the simulator.

2. Addition of extra fully connected layer for action in the Q-function to capture more non-linear behaviour of the agent.

3. Increasing the size of the experience buffer from $10,000$ to $100,000$ to make the states sampled in the mini-batch less dependent on the current policy.

4. Reduced high dropout between CNN and FCN parts of the network to reduce loss of information.

The CNN and FCN layers used in this have the same dimension as the baseline model and are described in the previous subsection. The modifications made to the architecture are as shown in the image below.

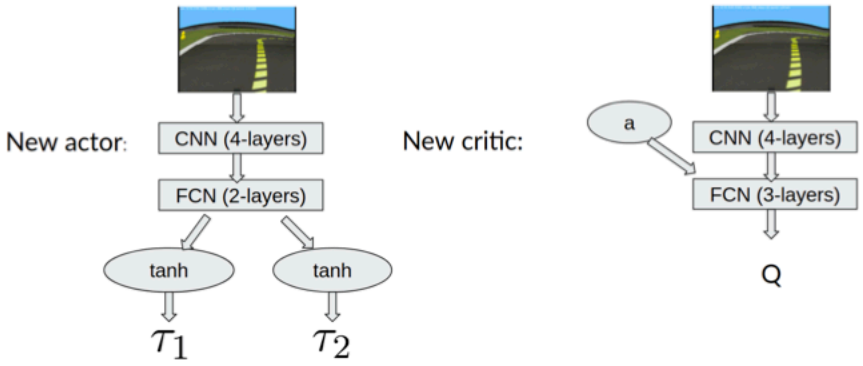


Figure 4: Modified Architecture

3.3.2 Encoding state with Convolutional Autoencoder

The baseline model performs an end to end training where the the state is a RGB image of size $3 \times 160 \times 120$. This means the state space lies in R^{57600} ($3 \times 120 \times 160 = 57,600$), which is HUGE! It will take the algorithm a lot of time steps to search over such a big state space and the chances are it might not be able to search all states in the first place. This makes exploring the state space difficult and lowers our chances of finding an optimal policy.

Since nearby pixels in an image are highly correlated and images are known to be sparse signals, we decided to reduce the state space by encoding the the information into a smaller state space of size R^{480}. Our proposed architecture is as shown in Figure 5 where the FCNs have 400 neurons in the first hidden layer and 300 neurons in the second hidden layer for both the actor and critic. Now whenever a state s is generated by the environment, it first goes through the encoder whose output $E(s)$ is used as the input to the Actor and Critic networks. Comparing this architecture to the one in Figure 4 where each of the actor and critic have their own state encoder (the 4 layered CNN), both actor and critic now share an encoder which is trained before hand.

To reiterate from the previous paragraph, our state is now $E(s) \in R^{480}$. Searching/Exploring a state space which is 120 times smaller than the original increases our chances of estimating good Q values for each state and hence finding a better policy. The encoding is learnt by means of a convolutional auto encoder (CAE) whose architecture is shown in Table 2. After training, we will only use the encoder part of the network represented by $E(.)$ whose input will be the state $s \in R^{57600}$ and output will be $E(s) \in R^{480}$

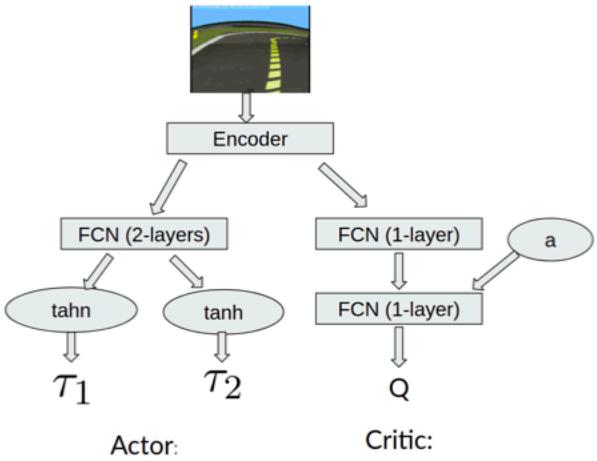


Figure 5: Architecture of actor and critic with convolutional encoder

To train the CAE, we created a data set of 40,000 and 10,000 training and validation images respectively. This was done by making the duckiebot (the agent) take random actions in duckietown (the environment) for 50,000 time steps. To make sure that our agent views almost all scenarios, we set its position to a random point in the environment every 10 time steps.

Layer	In	Out	Kernel	Padding	Stride
Encoder block $E(.)$:					
Conv, BatchNorm	3	8	3x3	1	1
Relu, Maxpool	8	8	2x2	0	1
Conv, BatchNorm	8	16	3x3	1	1
Relu, Maxpool	16	16	2x2	0	1
Conv, BatchNorm	16	16	3x3	1	1
Relu, Maxpool	16	16	2x2	0	1
Conv, BatchNorm	16	32	3x3	1	1
Relu, Maxpool	32	32	2x2	0	1
Conv, BatchNorm	32	32	3x3	1	1
Relu, Maxpool	32	32	2x2	0	1
Decoder block $D(.)$:					
MaxUnpool	32	32	2x2	0	1
Conv, BatchNorm	32	32	3x3	1	1
MaxUnpool	32	32	2x2	0	1
Conv, BatchNorm	32	16	3x3	1	1
MaxUnpool	16	16	2x2	0	1
Conv, BatchNorm	16	16	3x3	1	1
MaxUnpool	16	8	2x2	0	1
Conv, BatchNorm	8	8	3x3	1	1
MaxUnpool	8	8	2x2	0	1
Conv, BatchNorm	8	3	3x3	1	1

Table 2: Convolutional Autoencoder architecture

3.4 Change in Algorithm: TD3

TD3 stands for Twin Delayed DDPG. While DDPG is able to perform well, it is very dependent on its hyperparameters and needs tuning. Often, DDPG overestimates the Q-values. It is a variation of DDPG where it instead of one, it uses two target Q networks to prevent overestimation of Q values. This prevents error from propagating and helps stabilize the Q values which in turn helps learn a better policy. This is done by implementing the following steps:

1. Initialize two random target Q networks parameterized by $(\theta^{Q^1}, \theta^{Q^2})$ and one random policy network parameterized θ^{π^1}
2. Follow same steps as DDPG (create replay buffer, get an action from policy, etc) except we use the minimum of (Q_1, Q_2) to update the weights of the both critic networks $(\theta^{Q^1}, \theta^{Q^2})$
3. TD3 adds noise to the target actions, to make the policy explore better
4. Update target networks as usual

3.5 Change in Reward Function

We modify the original reward function as seen in equation (1) to motivate the Duckiebot to move forward. The modified reward function can be seen in equation (7). While the action space was designed to only allow forward movement (non-negative torques), this still allowed for the ability for the Duckiebot to spin in place, fully able to maintain a good reward. The staying-in-lane incentive is satisfied and the collision penalty is avoided when the Duckiebot spins. By adding the T_t^i term, we are incentivizing the car to get to new tiles by penalizing it linearly with respect to time for being in the same tile.Tile penalty contains two terms $R > 0$ a positive constant N_{t-1}^i is the number of

times tile i is visited till time $t-1$. The tile penalty is weighted like the other terms. We used 1, 0.2, -10, 40, 500 for $\alpha, \delta, \beta, \gamma, R$ respectively.

$$r_t = \alpha \vec{v_t}.\vec{d_t} + \delta T_t^i - \beta l_t - \gamma C_t \tag{6}$$

$$T_t^i = \begin{cases} R, & \text{if i is a new tile} \\ -N_{t-1}^i, & \text{otherwise} \end{cases} \tag{7}$$

3.6 Metrics

The total reward R is used as the metric. We do not include loss of the policy/value function as metrics because lower loss values doesn't imply a better learnt function.

4 Results and Discussion

4.1 Baseline: DDPG

The initial baseline was trained using the DDPG code provided in the Duckietown repository[5] and we got the following results. The graphs represent average rewards obtained over an episode by evaluating the obtained policy for 5 episodes, at regular step intervals. The result is noisy and the value does not converge due to incorrect description of action space for the baseline policy model. The modified DDPG network with changes described in section 3.3.1 where changes are made to accommodate for action space containing wheel torques, increased buffer size and improved dropout parameters performs better. The results obtained for these two settings can be seen in the figures 6 and 8.

It can be seen that the modifications work as expected. The reward over an episode converges to an optimal value of 1312 as compared to an average of 923. And the variation in the reward also decreases due to the stabilization of the policy. It is seen from the videos while testing the trained policy that the agent starts spinning around in circles with some small translation for baseline DDPG. But for the modified DDPG it optimises the policy by reaching the minima where it spins around in the same place, with 0 translation in the centre of the lane. This can be observed to be because of the reward function, where the robot gets penalised for going away from the centre of the lane. This behavior also gets reflected in the resultant graphs for rewards obtained over an episode. The variation to architecture to get faster convergence and better average rewards.

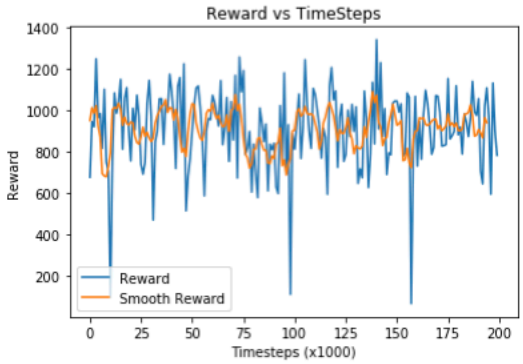
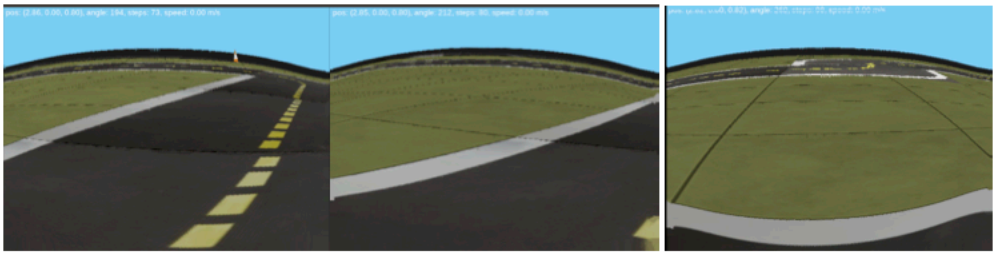


Figure 6: Rewards with initial DDPG architecture. Smooth reward represents the running average computed using a window of size 5.

Three frames of a test using the baseline DDPG model are as given below. It can be seen that the agent rotates with some small translation.



(a) Frame 1 (b) Frame 2 (c) Frame 3

Figure 7: Three frames of a test using the baseline DDPG model

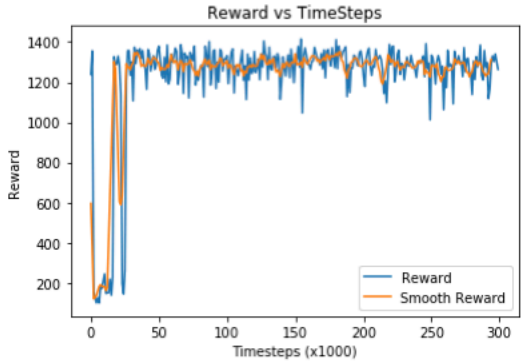


Figure 8: Rewards with our DDPG architecture. Smooth reward represents the running average computed using a window of size 5.

4.2 Change in Architecture: Encoding state with Convolutional Autoencoder

After training the CAE for 15 epochs with MSE loss, Adam optimizer with learning rate = 2×10^{-4}, we get the following loss curves as shown in Figure 9. We can clearly see that the loss on both train and validation set quickly drops and almost nears 0. This tells us that the CAE was able to successfully encode the information in R^{480} and reduce dimensionality of the state space by 120 times.

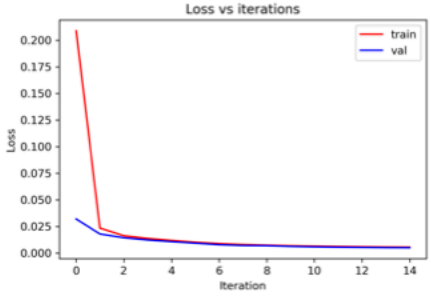


Figure 9: Convolutional Autoencoder loss curves

After training the encoder $E(.)$, we used the architecture shown in Figure 5 to train the agent with DDPG which resulted in the reward graph shown in Figure 10. As the number of timesteps increases, the reward increases and plateaus. This tells two things - 1) encoding the state can make RL algorithms work without sacrificing on the average reward yielded and 2) the original state has redundant information in it and encoding the state into a smaller space will work. Hence, our claim

is verified. Comparing this to the reward graphs of DPPG and TD3 in Figure 8 and 11, we can see that the reward plateaus at the same value. Hence, using an encoder led to the same average reward with the added advantage of searching a smaller state space.

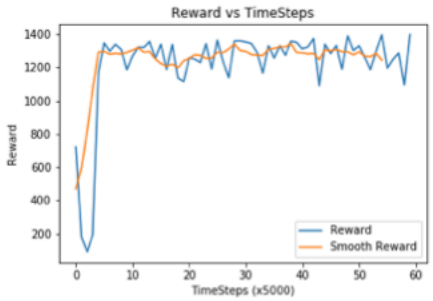


Figure 10: Rewards for DDPG with encoded states

4.3 Change in Algorithm: TD3

The results on training the Duckie Town with our TD3 algorithm is shown below:

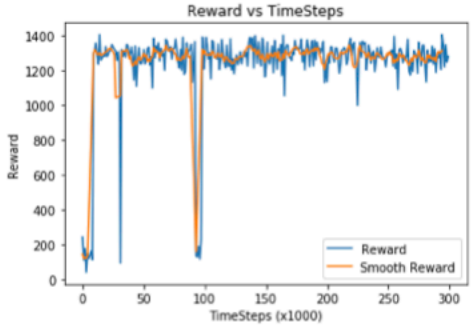
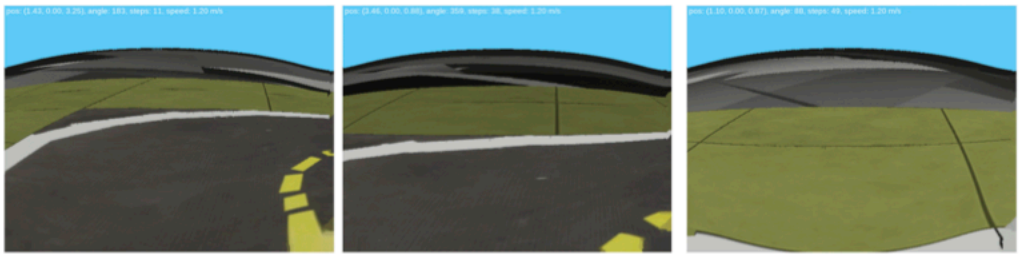


Figure 11: Rewards with TD3. Smooth reward represents the running average computed using a window of size 5.

As seen in the image above, the graph for TD3 converges faster, as expected and converges to a value of 1325. It has lower variation in the reward per episode for most of the period, but diverges at some points. The performance of the policy is also similar to that of the DDPG agent as they converge to the same minima. Hence the agent learns to spin at the same point while staying at the centre of the lane. To overcome this, we modify the reward function and experiment with the constants to get the best possible results.

4.4 Change in Reward Function

With the modified reward function, TD3 was able to train a policy that drove in a straight line. You can find frames of a test case where this is happening in figure 13. Because of the nature of the environment, the Duckiebot stabilized its rewards by ending the series of trajectories as soon as possible and leaving the lane. This way it would not be penalized for being in the same tile while still maintaining a positive reward. Because the tile penalization was higher than the staying-in-lane reward, it was better for the expected reward to end the run by leaving the lane. The training reward curve is seen in 12. The magnitude of reward for this variation is different from others because we edited the reward function. The final average reward had a magnitude of 309.



(a) Frame 1 (b) Frame 2 (c) Frame 3

Figure 13: Three frames of a test using the modified reward TD3 policy

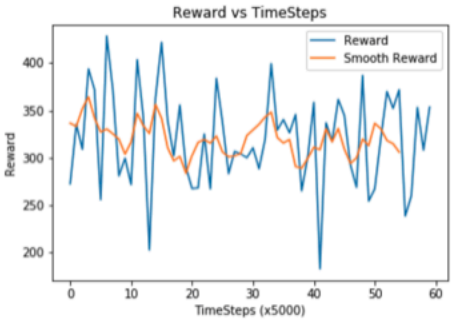


Figure 12: Reward curve with TD3 and a modified reward function

5 Conclusion

We successfully performed end-to-end learning of policy and Q-networks to maximise rewards. We tried out different architectures for DDPG, implemented TD3 and modified the reward function in order to show the importance of the reward function, and how varying it varies the behavior of the agent drastically. We successfully got the duckiebot to drive in a straight line in an environment with other duckiebots in different lanes.

6 Team Member Contributions

[redacted]

Set up docker image and script to allow usage of the simulator and dummy displays on the UCSD pods (the displays are necessary for simulator to work). Coded TD3. Worked on improving the results of DDPG/TD3 training and the modified reward function. For the report, wrote the abstract, introduction, motivation, reward function related sections, and the TD3 section.

[redacted]

Regarding code, I worked completely on the convolutional auto encoder (except for data generation which was done by [redacted]) and helped with setting up the duckietown environment. Regarding the report, I contributed to section 3.1.1 (Method: The RL problem), 3.3 (Method: TD3), 3.4 (Method: Encoding with CAE), 4.3 (Results: Encoding with CAE)

[redacted]

Worked on debugging the code. Generation of image buffer for CAE, Worked on changing architecture for DDPG, Modifying the code to use train encoder with fully connected network for actor

and critic, worked on modifying and tuning the reward function. For the report wrote the changed architecture, generated architecture models, result gifs and modified reward results. Worked on the discussion section and conclusion. Did the final comprehensive edits to the report.

[redacted]

Worked on seeing if the DDPG code worked with another environment, to double check that everything was correct. Also, worked on writing up the parts about RL preliminaries and DDPG on the report.

[redacted]

Worked on generating the results for baseline architecture and our change in the architecture for DDPG. Regarding the report, contributed to the same. Also, helped in debugging, organizing run and visualization of results for our change in architecture of the environment.

References

[1] Qi Zhang, Tao Du, and Changzheng Tian. Self-driving scale car trained by deep reinforcement learning, 2019.

[2] Matt Vitelli and Aran Nayebi. Carma : A deep reinforcement learning approach to autonomous driving. 2016.

[3] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. CARLA: An open urban driving simulator. In *Proceedings of the 1st Annual Conference on Robot Learning*, pages 1–16, 2017.

[4] Voyage deepdrive. https://github.com/deepdrive/deepdrive, 2019.

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

[6] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Manfred Otto Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *CoRR*, abs/1509.02971, 2015.

[7] Scott Fujimoto, Herke van Hoof, and David Meger. Addressing function approximation error in actor-critic methods. *CoRR*, abs/1802.09477, 2018.

Driving in Duckietown using Reinforcement Learning

By:

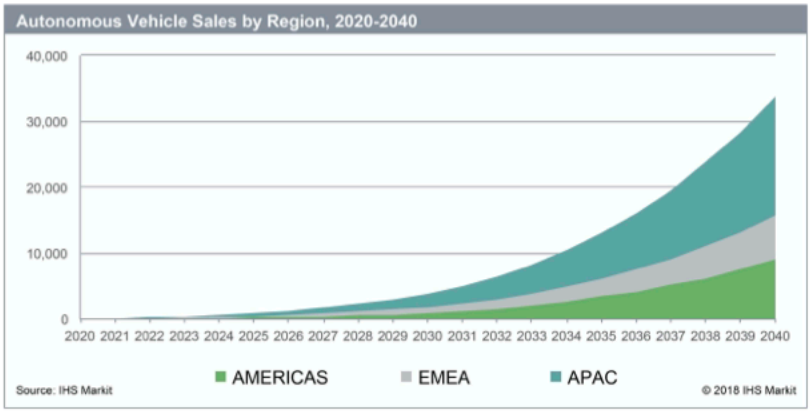
[redacted]

Problem / Technique

- Problem:
  - Driving around a simulated town using an artificially intelligent agent, trained using deterministic policy gradient algorithms.
- Techniques used:
  - DDPG - Deep Deterministic Policy Gradient
  - TD3 - Twin Delayed DDPG

Motivation

- Advancement:
  - End to End (pixel to control) learning of autonomous driving for optimal control and behavior of the agent.



- Aspects that make a difference:
  - Architecture of the actor and critic networks.
  - Algorithm being used (Obviously!)
  - Reward Function.

RL Framework

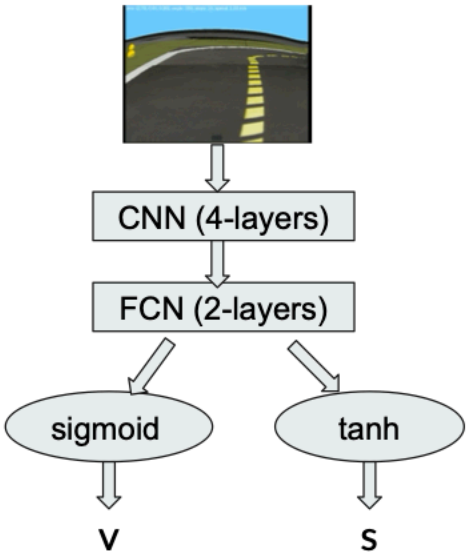
- **Agent:** Duckiebot, a differential drive robot.
- **Environment:** Duckie-town. This is a virtual town containing roads with intersections, obstacles, pedestrians, and other Duckiebots.
- **State (s):** 640 X 480 Image captured by the Agent
- **Action (a)**:
  - Setting1: (T1,T2) torque applied to left and right wheel respectively. Scaled to [0,1]
  - Setting2: (V,S) velocity of the vehicle and steering angle. Both scaled to [0,1]
- **Reward (r):** Rewards velocity(v) in the direction of the road(d) and penalises distance from centre of the lane(l) and vicinity to obstacles (C).
  - Formally : $r_t = \alpha \vec{v_t}.\vec{d_t} - \beta l_t - \gamma C_t$

Baseline: High level overview of DDPG

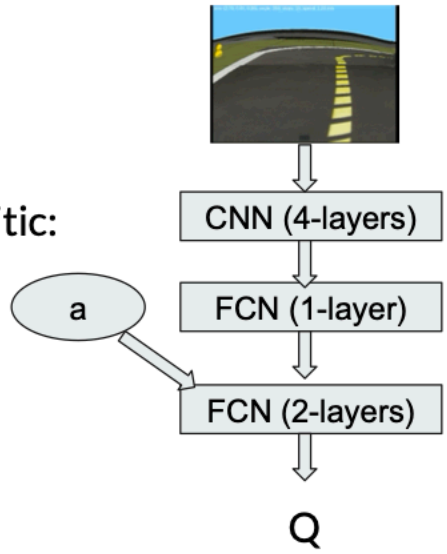
- Off-policy, deterministic policy gradient, actor-critic algorithm
- Two neural networks:
  - Policy Network: Determine action for the current state
  - Q-Network: Estimate long term reward for a state-action pair
- Key Ideas:
  - Prevent divergence by using ‘Target networks’
  - Use Replay buffer to break up time based correlations

Baseline architecture

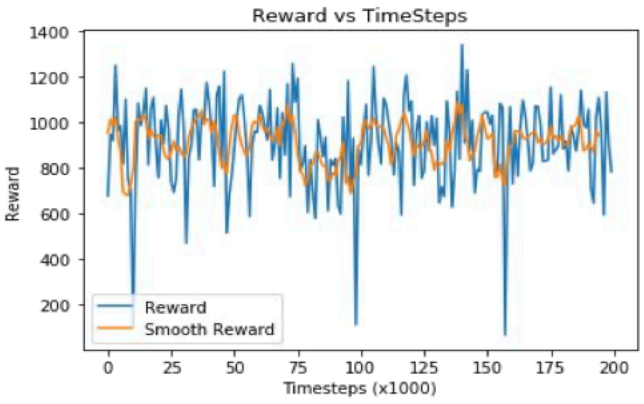
Actor:



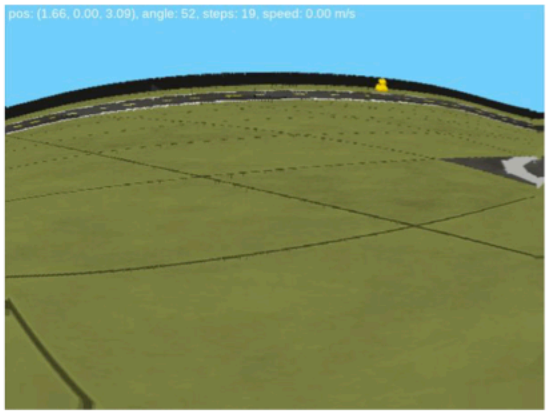
Critic:



Results - Initial DDPG



DDPG - reward graph

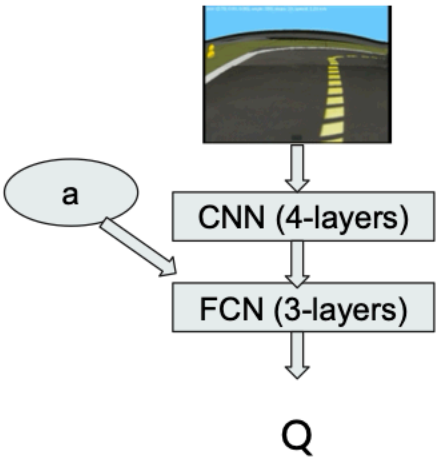


DDPG - GIF

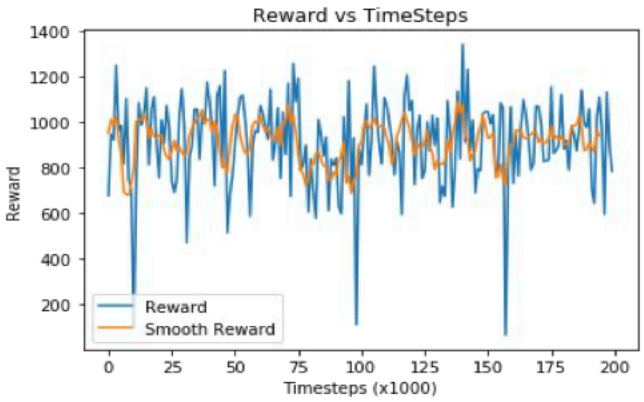
Change in architecture:

- Quality of life change: training on wheel torques
- Addition of extra fully connected layer for action in the Q-function
- Increasing the size of the experience buffer
- Removed high dropout between CNN and FCN parts of the network

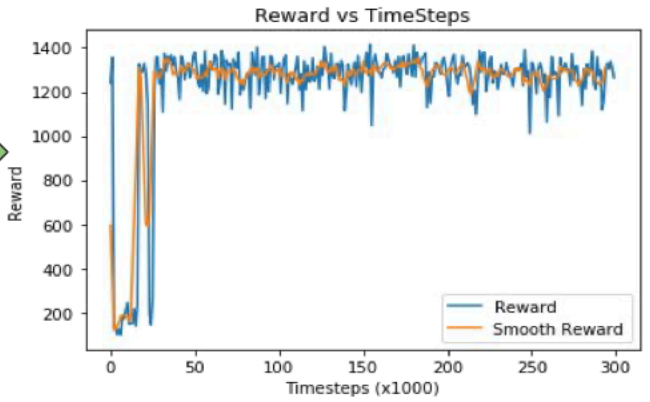
New critic:



Results - Graphs



DDPG - initial architecture

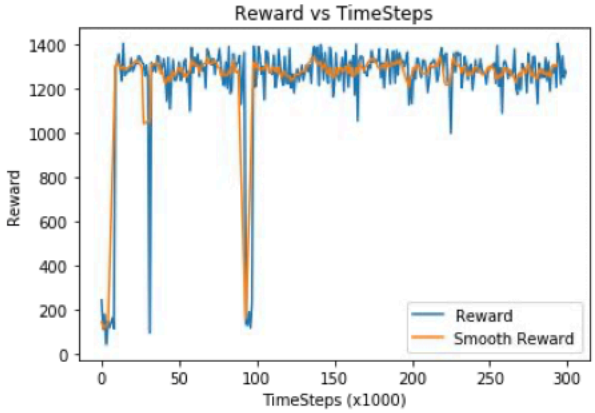


DDPG - our architecture

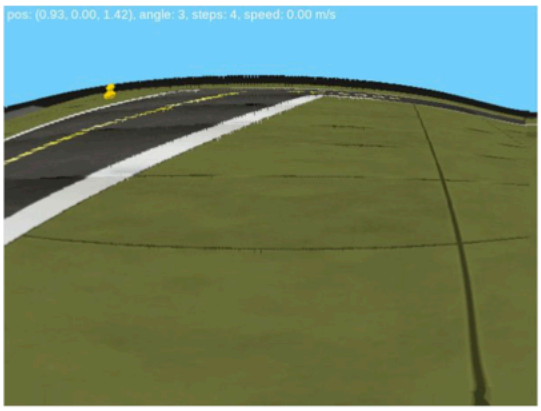
Change in algorithm: High level overview of TD3

- Variation of DDPG.
- Common failure in DDPG - the learned Q function overestimates Q values.
- TD3 addresses this issue by using three tricks:
  - TD3 learns two target Q functions instead of one and uses the minimum of the two values to update the weights.
  - TD3 updates policy and target networks less frequently than the Q-function.
  - TD3 adds noise to the target actions, to make the policy explore better.

Results: TD3



TD3



TD3 - GIF

Change in Rewards:

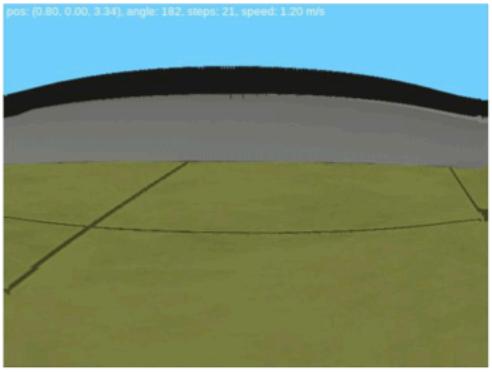
- The road is made using many tiles (regions).
- **Idea:** Reward term T is introduced which has a positive value(R) when the agent is on a new tile i. If tile i is not new, it has a value negative of the number of times the i has been seen(N) in the episode.

$$r_t = \alpha \vec{v_t}.\vec{d_t} + \delta T^i - \beta l_t - \gamma C_t$$

$$T_t^i = \begin{cases} R, & \text{if i is a new tile} \\ -N_{t-1}^i, & \text{otherwise} \end{cases}$$

Results: Modified Rewards

Modified Reward



Modified Reward - GIF

Discussion - How well did it work & why?

- The agent only learnt to drive in a straight line.
- Why:
  - **The reward function:** The relative importance among the terms in the reward function is not tuned enough to get desired behavior for the agent.
  - **Number of training episodes:** The number of time steps we train the agent for, is very important. 1M time steps is recommended, but we were able to train it only for 300K (took us 24+ hours!).
  - **Network architecture:** The proposed network architecture might not be good enough to capture the underlying complexity related to understanding when to turn.

Discussion - How can we improve it?

- How to improve results?
  - Find the right relative weights between all the terms in the reward function.
  - Train for more episodes.
  - Deeper networks for capturing better state representations from pixels and RNN/LSTM to pass information in time.

Summary

- Got a duckiebot to drive on a straight line in an environment with other duckiebots in different lanes.
- Experimented with the reward function, architecture and the buffer size to show how important they are in the case of deterministic policy gradient!

Thank you!

References

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

[2] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Manfred Otto Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning.CoRR, abs/1509.02971, 2015.

[3] Scott Fujimoto, Herke van Hoof, and David Meger. Addressing function approximation error in actor-critic methods.CoRR, abs/1802.09477, 2018.