The Road To Be Taken: A Deep Reinforcement Learning Approach Towards Autonomous Navigation

Bhanu Renukuntla, Saket Sharma, Swaroop Gadiyaram, Venkatesh Elango, Vikas Sakaray
Department of Electrical and Computer Engineering
University of California, San Diego

1 Introduction

In the past decade, there has been a growing interest in research to make autonomous driving a commercially-viable reality. A wide variety of approaches and methods have been discovered to this end. However, due to the increase in popularity of deep networks and their capabilities, they have become indispensable in the current state-of-the-art technology for perception tasks. These networks form the core AI that makes crucial decisions related to automobile navigation. Several current deep learning based approaches make use of Convoultional Neural Networks (CNNs) that require a lot of labeled data to train.

We approach this problem using a non-data centric Reinforced Learning (RL) framework. Reinforcement Learning algorithms are particularly well suited to consecutive decision-making tasks such as driving. This is achieved by using a model that explicitly accounts for the future. The RL agent starts without any knowledge about the world and gradually learns a mapping from states to actions by trial and error. Problems like these are difficult because agents need to anticipate the future. In our case, we focus on a training an RL agent to emulate a race driver's actions to complete laps as fast as possible. This process involves a lot of decision making. Every time the agent makes a decision (*action*) it interacts with the *environment* and gets the feedback signal as *reward*.

For our task, we use Google DeepMind's Deep Deterministic Policy Gradient (DDPG) algorithm[2] which is based on the Actor-Critic architecture[5] to learn an RL agent. This is an application of deep Q-learning which casts the problem in the form of increasing a *value* function. At its core, DDPG is a policy gradient algorithm that uses a stochastic behavior policy for good exploration but estimates a deterministic target policy, which is much easier to learn. The RL agent that takes sensor parameters as input. For example, from the environment engine critical indicators for driving, such as speed of the host car, the host cars relative position to the roads central line, the distance to the preceding cars etc which are generated in TORCS (The Open Racing Car Simulator) which has been used extensively in the AI community for research. No visual information is used by the agent. The *action* taken by the RL agent consists of three continuous real values - one for steering, one for braking and the other for acceleration. Both the actor and the critic are approximated using neural networks which are trained using back propagation. The actor network calculates the greedy action, to which we add some noise using the Ornstein Uhlenbeck (OU) process which aids in exploration of parameters. We propose and test a model that uses a *reward* which is a function of the car's velocity, the distance from the center of the track and the angle between car's longitudinal axis and track direction, and try a variety of experiments as well. To test our network, we check performance of our trained agent on an unseen track in TORCS.

2 Background

A lot of research has gone into autonomous driving in the last decade. Given the rise in popularity of deep learning based models a lot of researchers have tried to solve this problem from a deep learning perspective. Volodymyr Mnih, et al [8] presented a deep learning based approach to Re-

inforcement Learning called *Deep Q learning* to successfully learn control policies directly from high-dimensional sensory input using Reinforcement Learning. They used a Convolutional Neural Network, trained with a variant of Q-learning, that took raw pixels as input and generated a value function estimating future rewards at the output. However this method was suitable for problems involving a discrete set of actions.

A naive discretization of the action space throws away valuable information concerning the geometry of the action domain, while fine discretization leads to the curse of dimensionality problem. David Silver, et al[4] researched deterministic policy gradient techniques for reinforcement learning with continuous actions. This method was found to be much more efficient than the existing stochastic policy gradient techniques. They also introduced an off policy actor-critic algorithm that learnt a deterministic policy from an exploratory behavior policy. They reported significant improvements over stochastic policy gradient in high dimensional spaces.

Lillicrap, et al[9] from Google Deepmind tried to adapt the ideas of *deep Q-learning* to David Silver's approach. A model-free, off-policy actor-critic algorithm that uses deep function approximators to learn policies in high-dimensional, continuous action spaces was proposed. The proposed algorithm to tackle the continuous action space problem combined 3 techniques together: Deterministic Policy-Gradient Algorithms, Actor-Critic Methods[5], and a Deep Q-Network (called Deep Deterministic Policy Gradients (DDPG)). Our work is based off the work in this paper, with additional modifications discussed in following sections.

3 Model

While our initial plan was to use a CNN based approach to solve the autonomous navigation task we ran into certain memory leak issues on TORCS using large dataset (50 GB), an issue that's been reported in the past. We had also considered casting this as an RL problem. With repeated failed attempts to get past the bug in TORCS and limited time at hand, we decided to move ahead with the RL approach.

The Deep RL model uses separate vanilla neural networks for Actor and Critic modules. Actor network takes state variables (for example, distance of the car from track center, speed of the car, angle of the car with the track etc. which are captured from the environment using sensors) as inputs and gives acceleration, brake and steering as outputs. This is illustrated in Figure 1(a). Acceleration and brake outputs must lie within (0,1) range and to account for that the corresponding output nodes are sigmoid activated. For similar reasons, tanh activation function is used for steering node to limit the range to (-1,1). The network has 300 and 600 hidden nodes with ReLu activations for first and second hidden layers respectively.

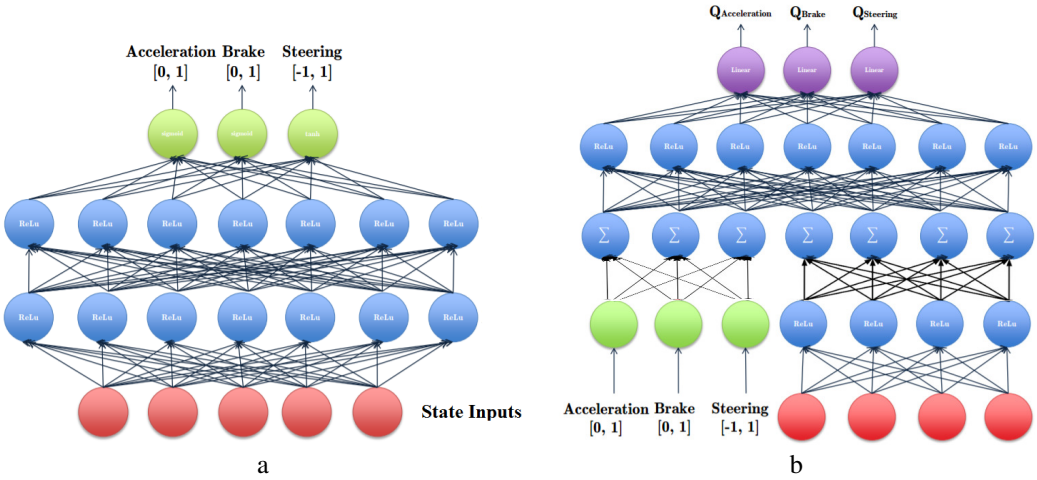

Figure 1: Illustration of (a) Actor Network (b) Critic Network

Architecture of Critic network is shown in Figure 1(b). Critic Network takes Acceleration, Brake, Steering (produced by the actor network) and State variables as inputs. The base network has 300 and 600 hidden nodes with ReLu activations for first and third hidden layers respectively. Second hidden layer has 300 linear nodes. Modified networks are used for a separate experiment (Sec. 5). Q values produced by the critic network are used in loss computations, shown in Section 3.2. As described in section 3.2, the actor network uses Q values computed from the critic network and learns to maximize expected discounted reward using policy gradient approach. The critic network uses continuous Q-learning to optimize the loss function based on the current state and action. Deterministic Policy Gradient gives the update rule for the weights of the actor network. The critic network is updated from the gradients obtained from the TD error signal.

3.1 Actor-Critic Methods

Actor-critic methods are Temporal difference (TD) methods that have a separate memory structure to explicitly represent the policy independent of the value function. The policy structure is known as the actor, because it is used to select actions, and the estimated value function is known as the critic, because it criticizes the actions made by the actor. Learning is always on-policy: the critic must learn about and critique whatever policy is currently being followed by the actor. The critique takes the form of a TD error. This scalar signal is the sole output of the critic and drives all learning in both actor and critic, as suggested by Figure 2.

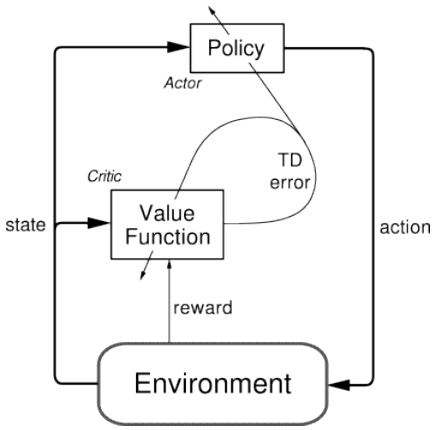

Figure 2: Sutton's Actor-Critic architecture [5]

Actor-critic methods are the natural extension of the idea of reinforcement comparison methods to TD learning and to the full reinforcement learning problem. Typically, the critic is a state-value function. After each action selection, the critic evaluates the new state to determine whether things have gone better or worse than expected.

Actor-critic methods are likely to remain of current interest because of two significant apparent advantages:

1. They require minimal computation in order to select actions. Consider a case where there are an infinite number of possible actions–for example, a continuous-valued action. Any method learning just action values must search through this infinite set in order to pick an action. If the policy is explicitly stored, then this extensive computation may not be needed for each action selection.

2. They can learn an explicitly stochastic policy; that is, they can learn the optimal probabilities of selecting various actions.

3.2 Policy Gradient for Continuous Action Space

Here we are using Policy-Based Reinforcement Learning and directly parameterize the policy.

$$\pi_\theta(s, a) = P[a|s, \theta]$$

where s is the state, a is the action and θ represents the model parameters of the policy network. We can think of policy as a function that captures the agent's behavior, i.e. a map from state to action. Actor network is essentially modeling a policy to control the car and θ are the weights of the neural network. The main objective is to find a policy $\pi_\theta(s, a)$ using Reinforcement Learning approach. We can define a cumulative discounted reward is defined as follows:

$$R = r_1 + \gamma r_1 + \gamma^2 r_2 ... + \gamma^n r_n$$

An obvious objective function for identifying a policy is the expectation of the Cumulative discounted reward

$$L(\theta) = E[r_1 + \gamma r_2 + \gamma^2 r_3 + ...|\pi_\theta(s, a)]$$

or

$$L(\theta) = E_{x\sim\ p(x|\theta)}[R]$$

where the expectations of the total reward R is calculated under some probability distribution $p(x \mid \theta)$ parameterized by some θ.

For a deterministic policy, $a = \mu(s)$, and using $Q(s_t, a_t) = R_{t+1}$ we can write the gradients as follows

$$\frac{\partial L(\theta)}{\partial \theta} = E_{x\sim\ p(x|\theta)}[\frac{\partial Q}{\partial \theta}]$$

Simplifying,

$$\frac{\partial L(\theta)}{\partial \theta} = E_{x\sim\ p(x|\theta)}[\frac{\partial Q^\theta(s, a)}{\partial a}\frac{\partial a}{\partial \theta}]$$

We approximate Q-function as a neural network, $Q^\theta(s, a) \approx Q(s, a, w)$, where w is the weight of the neural network. Therefore, we arrive at the Deep Deterministic Policy Gradient formula:

$$\frac{\partial L(\theta)}{\partial \theta} = \frac{\partial Q(s, a, w)}{\partial a}\frac{\partial a}{\partial \theta}$$

where the policy parameters θ can be updated via stochastic gradient ascent.

We iteratively update the Q-function (Critic Network) to minimize the Loss function given by

$$Loss = [r + \gamma Q(s^{'}, a^{'}) - Q(s, a)]^2$$

3.3 Exploration Strategy

Exploration is done by adding noise to the actions predicted by the actor. We use Ornstein-Uhlenbeck process because of its mean reverting properties – which can be described by the following equation

$$dx_t = \theta(\mu - x_t)dt + \sigma dW_t$$

here W_t denotes the Wiener process and θ symbolizes how "fast" the variable reverts towards the mean. μ represents the equilibrium or mean value. σ is the degree of volatility of the process. While training, we gradually decay the noise over about 100k time steps.

3.4 Reward functions

We define

1. V_x as the speed of the car along it's longitudinal axis
2. V_y as the speed of the car along it's transverse axis
3. θ as the angle between the car's longitudinal axis and track direction

4. $trackpos$ as the distance between the car and the track axis normalized by the width of the track.

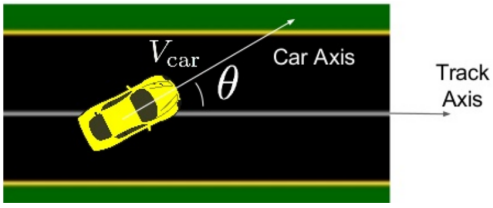

Figure 3: Car position with respect to track

The rewards that we consider are given below.

Baseline (DeepMind):

$$r = V_x cos\theta \tag{1}$$

Modified DeepMind:

$$r = V_x cos\theta - V_x|sin\theta| - V_x|trackpos| \tag{2}$$

Proposed Reward:

$$r = V_x cos\theta - V_x|sin\theta| - V_x|trackpos||sin\theta| - V_y cos\theta \tag{3}$$

Equation 1 is not expected to work with great success for TORCS as mentioned in [9].

4 Experiments & Results

We carry out a variety of experiments using the network described in the previous section. To make comparisons reasonable we use the same input parameters for all the models and train them on the same race track (*e-track-2*) as shown in Figure 7. Also, we test the models on the same track (*Alpine*) as shown in Figure 8. The test track is of a longer length and has more turns compared to the training track. In other words, it is a more 'difficult' track to race on. The reason why we selected this for testing was because we wanted to ensure that the model did not overfit to the training track, and that it generalizes well.

We use the original settings from [9] and [2] for training their models respectively. For the proposed model, however, we incorporate the following:

1. We incorporate a new reward function as given in Equation 3.
2. We use stochastic braking while training – the idea is to allow the car to brake 10% of the time during the exploration phase. The remaining 90% of the time we donǵ let the car brake which lets it learn how to accelerate.
3. We use Soft off-road Penalty – the idea is to allow the car to wander away from the center of the track but penalize the reward the longer it stays there. Also, we don't do this everytime the car moves away to the edge of the track but only with a probability of 10% (StopProb). With a probability of 90%, we restart the car's position.

For the sake of clarity, the experiments are discussed in more detail in the next section. We discuss the plots here.

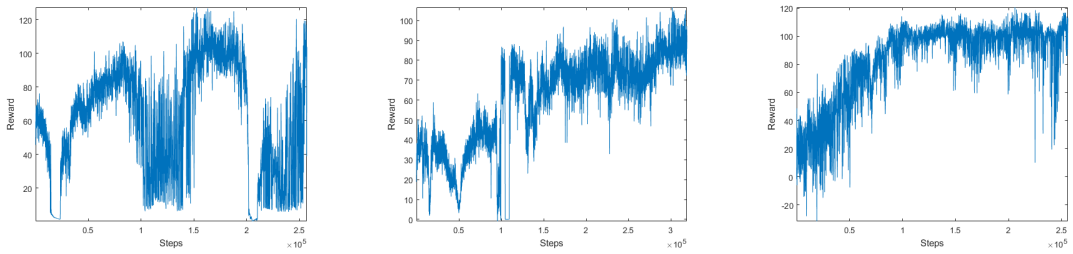

Figure 4: Reward obtained in training using different models (a) Google DeepMind (b) Modified DeepMind (c) Proposed Model

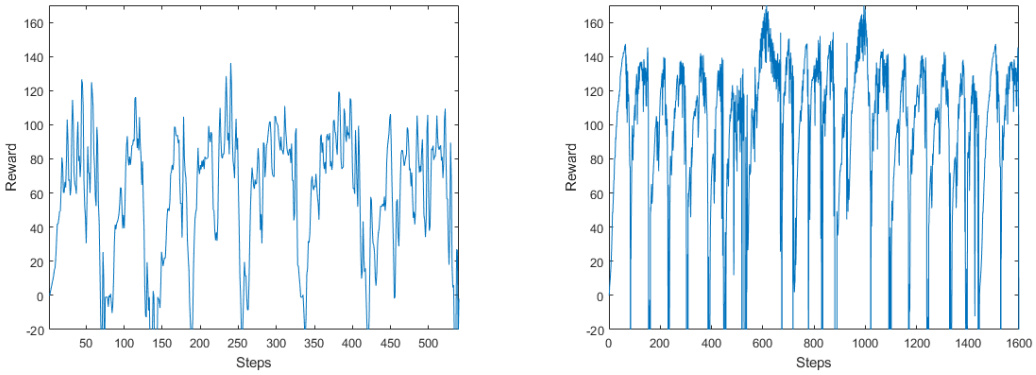

Figure 5: Reward obtained in testing using different models (a) Modified DeepMind (b) Proposed Model

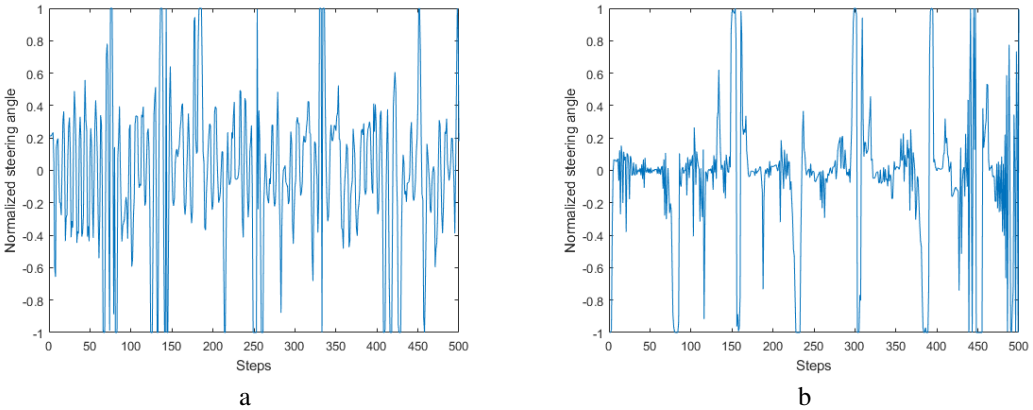

Figure 6: Normalized steering angle in testing using different models (a) Modified DeepMind (b) Proposed Model

From Figure 4, we see that the reward (running average over 200 steps) during training is highest for the proposed model. This is even after we added 2 extra penalizing terms to the reward while training. The plot is smoother and is a clear improvement over the method that uses Modified DeepMind reward. As mentioned by DeepMind their chosen reward performs poorly for TORCS.

The RL agent trained using DeepMind's reward was not able to drive on the test track and stopped part way through the testing. Thus we don't show the results from the DeepMind model in the testing track. The reward of the proposed model is comparatively higher than that of Modified DeepMind. The valleys in Fig 5 (b) are due to the car venturing into the flanks of the race track. The RL agent immediately takes the right action, swerving back to stay on track thereby increasing the reward. Note that unlike the previous plot, the reward here is not a running average. The RL agent trained using Modified DeepMind failed to complete an entire lap and hence has fewer steps. On the other hand, our model successfully completed an entire lap without crashing.

From Figure 6, we see that the proposed model learns a more stable driving style not having to change direction very often which can lead to jerky motion and reduced lap timing. The modified DeepMind model seems to change the driving direction very often – which is not desirable. We quantify the stability of the agent by computing the variance of the steering angle in test track. We obtained a variance of *0.196* and *0.169* with the modified DeepMind model and our proposed model respectively.

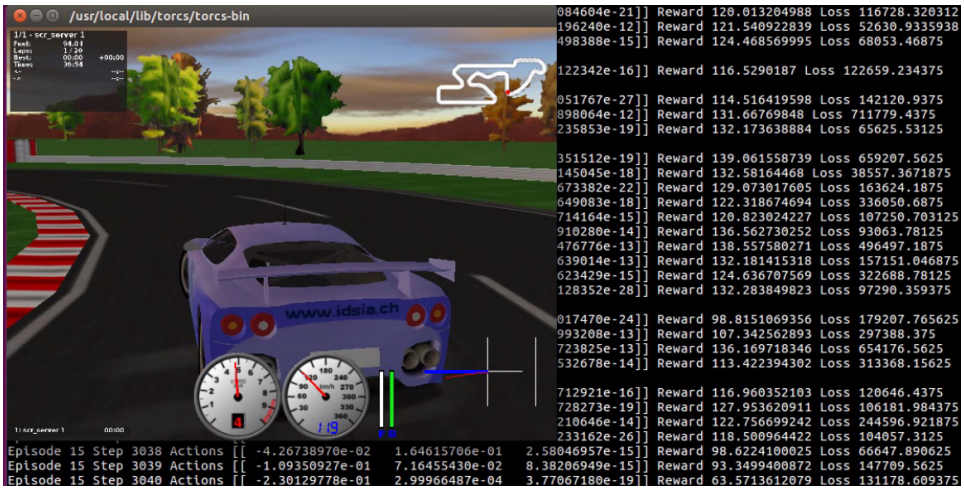

Figure 7: Screen grab showing our agent navigating in the training track

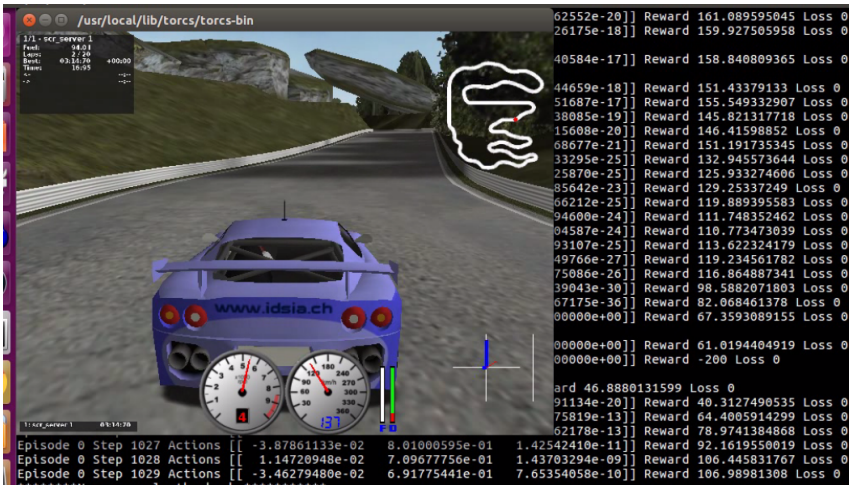

Figure 8: Screen grab showing our agent navigating in the testing track

5 Discussion and Conclusions

5.1 Reward function

By using Modified DeepMind reward function(eq. 2), the trained model appeared to change the driving direction quite often even on straight stretches where it's not necessary. This does not serve our purpose as it can slow down the lap time and makes the driving unstable. Though there could be multiple reasons for this, we wanted to see if we could change the reward function to get around this problem. One idea that seemed intuitive to us was to replace $V_x cos(\theta)$ with $V_x cos(2\theta)$. This change would not only make the model avoid turns greater than 45 degrees, but also make the car stay inline with the track direction.

However, experiments showed that the resulting model failed to learn to make proper turns and often spun the car out of track. Though the result was what we were not expecting, we figured why this approach failed. We believe that our suggested modification focuses too much on staying in the center of the race track. So, the car picks up a lot of speed before entering a turn and consequently goes out of control at turns. As a result the model is not stable.

One of the first observations we made was that the formulation of Google DeepMind reward isn't ideal – the velocity perpendicular to the body of the car isn't being penalized.We modified the reward to penalize $V_y cos(\theta)$ as well as $V_x|sin(\theta) trackpos|$ (Equation 3).

5.2 Stochastic Braking

Braking is a crucial action associated with driving and it turns out that teaching a model to brake is much harder than steering or acceleration. The reason for this is straightforward and has to do with the reward function. Since the reward is directly related to the velocity of the car, slowing a car down means a reduction in the reward. This makes the AI agent avoid braking as much as possible. Also, it is a bad idea to explore braking and acceleration simultaneously while training as the model can learn to brake harder and before it learns to accelerate well which can push it to a local minima (car coming to a halt and no reward is received). One way to get around this is to use stochastic braking. The idea is to allow the car to brake 10% of the time during the exploration phase. The remaining 90% of the time we don't let the car brake which lets it learn how to accelerate. The model avoids local minima while training, by allowing the car to acquire some non-zero velocity as well as learn how to brake at the same time.

5.3 Soft Off-road Penalty

By using Modified DeepMind reward function the model can learn a policy that keeps the car in the middle of the race track i.e, we prefer the car to not drive off center of the track. Though this policy yields reasonable results, the driving style/actions it learns isn't ideal for a race car. For instance if the turn in the track is smooth, race drivers cut in from wide as this allows them to maintain their speed and not have to brake much. On the other hand if the turn is sharp, it's is better to stay close to the inner curve of the road. We were hoping that given enough time, the model would learn doing these actions eventually just like how us humans learn with experience. However, the reward function used to train seemed too stringent to allow for this. Also at the time of training, [2] reinitializes the environment every time the car drifts away towards the edge of the race track. This results in a driving style that's particularly not well suited for race tracks. To overcome these limitations, we incorporate stochastic restarts while training.

The idea is to allow the car to wander away from the center of the track but penalize the reward the longer it stays there. Also, we don't do this everytime the car moves away to the edge of the track but only with a probability of 10% (StopProb). With a probability of 90%, we restart the car's position. This is a stochastic method different from [2] where the episode (refers to a start - crash sequence) is terminated immediately after the car goes off track. Initially we gave it a reward of -200 for every time step it remained outside the track. While this increased the number of steps per episode, it penalized the agent very heavily for leaving the track. As a result, we observed that after around 1200 episodes of training the agent stopped moving completely. Thus, we reduced the penalty to -20 and found that we weren't running into the cold start problem, thereby making the model more stable. Experimental results show that the new model successfully learns what we expected and

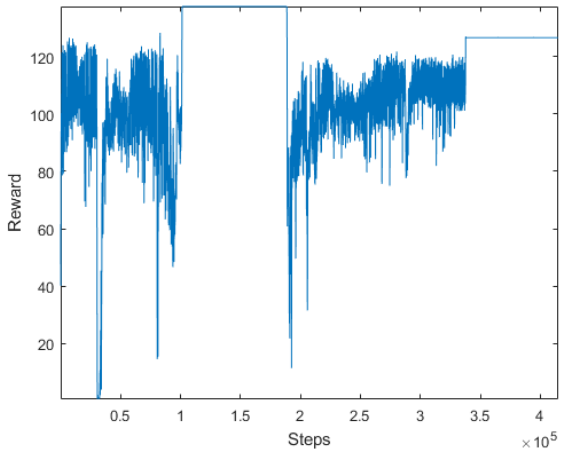

Figure 9: Continued training on Alborg track after training on e-track-2

closely replicates a race driver's behavior, and subsequently mitigates the issue we faced with our first experiment which made the car go out of control at bends.

Motivated by good results, we decided to incorporate *annealing* into our model. Specifically, we gradually decayed the value of StopProb over iterations. The intention was that this could possible lead to faster lap time by learning not to spend more than required time on the race track flanks. However, we couldn't find any notable difference (maybe a slight improvement in training speed but we couldn't quantify this) with a model that incorporated this. In sum, the stochastic restart procedure was found to be successful at achieving our goal and the annealing procedure isn't necessary as it offers no significant advantages over it.

5.4 Deeper Actor and Critic networks

Keeping the number of parameters fixed, we were interested to see if a deeper networks would learn a better state-action mapping and result in a more stable model (run multiple laps). The modified Actor network consisted of 3 hidden layers with 300, 500 and 80 nodes. For the Critic network, hidden layer consisting of 600 nodes was replaced with two layers made up of 360 nodes each. Just altering the Critic network led to an unstable policy. Reverting to the original Critic architecture and training with the modified Actor network, we observed the agent was able to learn a similar policy. The initial Actor architecture was good enough and increasing the depth doesn't help improve results. As for the Critic network, we believe that increasing the depth without any form of dropout leads to overfitting and causing the network to learn an unstable policy.

5.5 Training on multiple race tracks

Another interesting experiment we wanted to try was to train the model on multiple tracks (thereby allowing the agent gain sufficient 'driving experience') to see if this can lead to better generalizability. To do this, we had to carefully tune the OU process. If the agent is trained on the first track till the noise decayed to zero, then training on the second track with the OU process would render what it learned from the first track useless. In other words, we can get the same results by training on the second track alone. Our solution to this was to stop training on the first track before the noise completely decayed to zero, and continue training on the second track with the same level of noise at which we stopped training and so on. This makes exploration possible even after changing the tracks while training. Based on our experiments we found that tuning the hyper-parameter for the OU process is not straightforward. We are still working on finding a good method, but preliminary experiments indicate that this could lead to better generalizability as shown in Fig 9 where there are entire episodes in which the agent cruises along the track at steady velocity.

6 Concept and Innovation

In this project, we proposed a new RL agent, non data-centric approach for autonomous driving. The proposed method is better than the ones proposed in [9] and [2]. It learns to brake, make faster turns comparatively, completes laps, and generalizes well to new race tracks. These could be achieved using the following innovative procedures:

1. **Proposed Reward Model** - This led to faster and stable learning, indicated by higher reward values and smoother driving action.
2. **Stochastic Braking** - This technique helped the RL agent learn how to apply brakes which we believe is part of the essence of driving.
3. **Soft Off-road Penalty** - This technique allowed the RL agent to learn to make turns using the flanks of the road. As a result, the race car improved lap timing by not having to slow down more than required.
4. **Deeper Actor and Critic network** - While changing the Actor network gave similar results, altering Critic network led to an unstable policy.
5. **Training on multiple race tracks** - Generalizable RL driving agent able to learn very good policies.

As for analyzing and comparing the models, many attempts were made at coming up with plots and parameters that could help gain insight into their performance and understand them better. While we have included plots and results that capture the moving average of the reward, normalized steering angle as well as the variance of the steering angle, we also tried other analysis such as measuring jerk (derivative of the acceleration), capturing braking behavior etc. However, we couldn't draw meaningful inferences from these and decided to leave them out of this report.

References

[1] Chenyi Chen, Ari Seff, Alain Kornhauser and Jianxiong Xiao, *DeepDriving: Learning Affordance for Direct Perception in Autonomous Driving*, in Proceedings of 15th IEEE International Conference on Computer Vision, Santiago, Chile, 2015.

[2] Yan-Pan Lau, *Using keras and deep deterministic policy gradient to play torcs*, `https://yanpanlau.github.io/2016/10/11/Torcs-Keras.html`, 2016.

[3] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al, *Human-level control through deep reinforcement learning*, Nature, 518(7540):529533, 2015.

[4] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller, *Deterministic policy gradient algorithms*, in Proceedings of 31st International Conference of Machine Learning, Beijing, China, 2014.

[5] Richard S Sutton and Andrew G Barto, *Reinforcement learning: An introduction, volume 1*, MIT press Cambridge, 1998.

[6] Bernhard Wymann, E Espi, C Guionneau, C Dimitrakakis, R Coulom, and A Sumner, *Torcs, the open racing car simulator*. Software available at `http://torcs.sourceforge.net`, 2000.

[7] Daniele Loiacono, Luigi Cardamone, Pier Luca Lanzi, *Simulated Car Racing Championship: Competition Software Manual*, eprint arXiv:1304.1672

[8] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra and Martin Riedmiller, *Playing Atari with Deep Reinforcement Learning*, eprint arXiv: 1312.5602.

[9] 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*, ICLR 2016.

[10] Adithya Ganesh, Joe Charalel, Matthew Das Sarma, Nancy Xu, *Deep Reinforcement Learning for Simulated Autonomous Driving*, 2016.

7 Member Contributions

The contributions of all members are roughly equal. We worked on all aspects of the project together and collaboratively wrote the report.