Learning to Play TORCS using Deep-RL

Ayush K. RAI
MSc in AI
ayush.rai2512@student-cs.fr

Kai-Wei TSOU
OMA
kai-wei.tsou@supelec.fr

Benoit LAURES
OSY/MSc in AI
benoit.laures@student.ecp.fr

Kimya DHADE
MSc in DSBA
kimya.dhade@student.ecp.fr

Abstract

In this work, we explore deep-reinforcement learning methods like Deep Q-Learning[11], Double Deep Q-Learning[14] and Deep Deterministic Policy Gradient[9] for autonomous driving in The Open Racing Car Simulator (TORCS). Using the TensorFlow and Keras software frameworks, we train fully-connected deep neural networks that are able to autonomously drive across a diverse range of track geometries. A reward function aimed at maximizing longitudinal velocity while penalizing transverse velocity and divergence from the track center is used to train the agent. To validate learning process, we analyze the reward function parameters of the models over two validation tracks and qualitatively examine the driving stability. A video of the learned agent driving in TORCS can be found online[1].

1 Introduction

Our research objective is to apply deep reinforcement learning methods to train an agent that can autonomously race in TORCS (The Open Racing Car Simulator)[4]. TORCS is a simulation platform, which is used for research in autonomous driving and control systems. Accurate simulation platforms provide robust environments for training reinforcement learning models which can then be applied to real-world settings through transfer learning.

The field of Artificial Intelligence has made significant progress in being able to solve high-dimensional and complex sensory inputs. This has been made possible by combining (Krizhevsky et al., 2012) the theory behind Deep Learning and Reinforcement Learning thus creating Deep Reinforcement Learning algorithms. This aided in maneuvering the limitations posed by traditional reinforcement learning algorithms such as its inability to process complex high dimensional inputs. The "Deep Q Networks" (DQN) algorithm (Mnih et al., 2015) introduced soon after marked a milestone as it surpassed human expert in playing Atari game with no architecture adjustment or learning algorithms, only using raw unprocessed pixels as input.

The use of DQN, however, posed a challenge of dealing with continuous action space. This was a significant obstacle, since a vast majority of problems in robotic control and gaining advanced human level expertise in other applications fall into this category. On one hand discretizing continuous action space sparsely caused curse of dimensionality problem that is unfavourable for tasks that need finer control of action as a result finer grained discretization. On the other hand, a low level discretization of the action space throws away the valuable information available that concerns the geometry of the action domain.

[1] https://www.youtube.com/watch?v=Ok4Uv5jgUDs

Name	Description
ob.angle	Angle between the car direction and the track
ob.track	Vector of 19 range finder sensors: each sensor returns the distance between the track edge and the car within a range of 200 meters
ob.trackPos	Distance between the car and the track axis. The value is normalized w.r.t. to the track width: it is 0 when the car is on the axis, values greater than 1 or -1 means the car is outside of the track.
ob.speedX	Speed of the car along the longitudinal axis of the car (good velocity)
ob.speedY	Speed of the car along the transverse axis of the car
ob.speedZ	Speed of the car along the Z-axis of the car
ob.wheelSpinVel	Vector of 4 sensors representing the rotation speed of wheels
ob.rpm	Number of rotation per minute of the car engine

Table 1: The variables from the environment we input to the neural network.

In this project we apply Deep Q-Learning[11], Double Deep Q Learning[14] and Deep Deterministic Policy Gradient[9] and finally compare the results.

2 Task Description

In this project our goal is to learn a policy that can drive the car in the TORCS Racing Simulator[10] as fast as possible. In order to address this task, we apply Deep Q-Learning[11], Double Deep Q Learning[14] and Deep Deterministic Policy Gradient[9] on Input Sensor data from TORCS and in turn we predict acceleration, brake and steering angle. A reward function aimed at maximizing longitudinal velocity while penalizing transverse velocity and divergence from the track center is used to train the agent. To validate learning process, we analyze the reward function parameters of the models over two validation tracks and qualitatively examine the driving stability.

3 TORCS Environment

TORCS [1] is a multi-platform open-source 3D car racing simulator designed to enable pre-programmed AI drivers to race against one another, while allowing the user to control manually a vehicle. It features many different cars, tracks and physical features to resemble reality such as a simple damage model, collisions, tire, wheel properties (springs, dampers, stiffness...), aerodynamics (ground effect, spoilers...)... Python wrapper of TORCS called gym-TORCS [13] is available as a open source software for reinforcement learning research community.

Figure 1: TORCS Simulation Environment

The information provided by the environment [10] mimics those that would be collected by a real car or those traditionally given in video games: hence a state is defined by the output of 18 sensors (speed, angle, gear...). Regarding the actions, it is possible to control several elements, also close to what we are used to in video games (amount of acceleration, brake, steering...). Note that most values are continuous and on very different scales.

4 Related Work

There are two notable, distinct past approaches to training autonomous driving agents in TORCS. In 2015, the DeepDriving model applied deep supervised learning and convolutional neural networks (CNN) to learn a driving policy [3]. Other research, such as that done by DeepMind, focuses on reinforcement learning with CNNs [8]. The research by DeepMind demonstrates the wide applicability of actor-critic architectures, which use a pair of neural networks to address deep reinforcement learning, to continuous control problems. Further advances have made it possible to apply Deep Q-Learning to learn robust policies for continuous action spaces. The deep deterministic policy gradient (DDPG) method [8] presents an actor-critic, model-free algorithm based on the deterministic policy gradient. The algorithm is designed for physical control problems over high-dimensional, continuous state and action spaces. As such, it has potential applications in numerous physical control tasks. This paper continues to explore deep Q-learning for continuous control within the context of The Open Racing Car Simulator (TORCS).

5 Adopted Deep RL Methods

5.1 Deep Q-Network

Traditional Q-learning learns the action-value function $Q(s, a)$, which measure how good to take an action a given a state s. However, the drawback of Q-learning is that it sufers from high dimensional state and action. If the combinations of states and actions are too large, it is impossible to learn such Q-table. So researcher propose deep Q-network (DQN) [11], who uses a neural network to estimate the action value function. Given a certain state, the neural network aims to estimate the value corresponding to each action. In our project, we first tried to use DQN to learn to drive.

Note that since DQN can only deal with discrete action, we should first discretize our action space. In our TORCS simulator, we have three actions: steering, acceleration, and brake. The value for steering ranges from -1 (left) to 1 (right). The value for acceleration (resp. brake) ranges from 0 to 1, where 0 means no acceleration (resp. no brake) and 1 means full acceleration (resp. full brake). In our experiments, we quantized the action for steering into 30 levels, and 15 levels for steering and acceleration. We also adopted training tricks mentioned in the paper [11], including replay buffer and target network, in order to make training more stable. We use only 19 variables from the environment as our input to the neural network, which contains 2 hidden layers with 100 neurons.

Algorithm 1 DQN. We use θ to denote weights in primary network and θ' to denote weights in the target network.

Initialize replay memory D to capacity N
Initialize action-value function Q with random weights
for episode=1,M **do**
    Initialize sequence $s_1 = \{x_1\}$
    for t=1,T **do**
        With probability ϵ select a random action a_t
        otherwise select $a_t = \arg\max_a Q(s_t, a; \theta)$
        Execute action a_t and observe reward r_t and next state s_{t+1}
        Store (s_t, a_t, r_t, s_{t+1}) in D
        Sample minibatch from D
        if Game over **then**
            Set $y_j = r_j$
        if Not game over **then**
            Set $y_j = r_j + \gamma \max_{a'} Q(s_{t+1}, a'; \theta')$
        Update the neural network

5.2 Double Deep Q-Network

In [6], they have proved that traditional Q-learning suffers from over estimation problem, which might cause some negative effect on the performance. In double Q-learning, two value functions are

learned by randomly updating one of them. In each update, one value function is used to determine the greedy policy and the other to determine its value. The following two equations, which are the targets for the neural network, point our the difference between Q-learning and double Q-learning:

$$Y_t^Q = R_{t+1} + \gamma Q(S_{t+1}, \arg\max_a Q(S_{t+1}, a; \theta_t); \theta_t) \tag{1}$$

$$Y_t^{DoubleQ} = R_{t+1} + \gamma Q(S_{t+1}, \arg\max_a Q(S_{t+1}, a; \theta_t); \theta'_t) \tag{2}$$

Researcher further take advantage of both DQN and double Q-learning and tried to combine them, which results in Double DQN (DDQN) [14]. With two networks, it allows DDQN to estimate the reward more precisely. The algorithm in detail in shown in Algorithm 2.

Algorithm 2 DDQN. We use θ to denote weights in primary network and θ' to denote weights in the target network.

Initialize replay memory D to capacity N
Initialize action-value function Q with random weights
for episode=1,M **do**
    Initialize sequence $s_1 = \{x_1\}$
    for t=1,T **do**
        With probability ϵ select a random action a_t
        otherwise select $a_t = \arg\max_a Q(s_t, a; \theta)$
        Execute action a_t and observe reward r_t and next state s_{t+1}
        Store (s_t, a_t, r_t, s_{t+1}) in D
        Sample minibatch from D
        if Game over **then**
            Set $y_j = r_j$
        if Not game over **then**
            Set $a' = \arg\max_{a'} Q(s_{t+1}, a'; \theta)$
            Set $y_j = r_j + \gamma Q(s_{t+1}, a'; \theta')$
        Update the neural network

5.3 Deep Deterministic Policy Gradient Methods

Deep Deterministic Policy Gradient Methods (DDPG) [9] is a model using a neural network to predict the state value function of a Markov decision process problem, with continuous state and action spaces. It uses main ideas from DQN and theory in neural network to build a model capable of learning competitive policies for different continuous task such as the cartpole swing-up task or, in our problem, learning to drive in TORCS. Continuous problems are quite challenging in reinforcement learning because usual methods can't be used efficiently. For instance, as discussed earlier, DQN has proved to be very effective, achieving human level performance on Atari games from unprocessed pixels using a neural network to approximate the action-value function. However, it is only good for discrete or low dimensional action spaces because an action has to be sampled from it. Hence, to apply directly such a strategy in a continuous problem, it would be necessary to discretize first the action space which would be highly uneffective because, since at each step the network maximizes the Q function to compute the target label, maximizing such a function over a continuous space at each iteration would be computationally way too big.

DDPG is an off-policy algorithm but can't use the greedy policy, thus it sticks the same idea as DPG: using an actor function $\mu(s|\theta^\mu)$ that specifies the current deterministic policy (mapping states to actions). To approximate the Q function, it uses a neural network, thus introducing non-linearities. However, to train properly (and more quickly) a network, it is preferred to use mini-batch and non-correlated data which is a priori impossible in deep learning. To preserve convergence, it takes advantage of the same ideas as in DQN and adapt them:

- It uses a replay buffer (as large as possible to favor uncorrelated transitions) to store transitions as tuples (s_t, a_t, r_t, s_{t+1}), discarding the oldest ones when full and drawing from this buffer to train a mini-batch.

- In DQN, the "input" (transitions) and "output" (target values) are sampled both from the network using the same policy. This is prone to divergence, thus to increase stability, the authors make a copy of the trained network (μ', Q'), just like in DDQN, which is used to sample the target values and is updated by having him slowly track the learned networks. It slows down a bit the training but makes the whole pipeline much more robust.

- Finally, to improve training and since values in the reward vector often have a real meaning and different physical units, thus varying over very different scales. The authors just used the well known Batch Normalization technique to normalize the data and improve stability.

Finally, they found that an easy way for exploration in off-policy continuous problems is to add an exploration noise to the actor policy: $\mu'(s_t) = \mu(s_t|\theta_t^\mu) + \mathcal{N}$.

The interest of the model is that it is very efficient to learn policies and, in simple tasks, it shows even better performance by learning them directly from pixels rather than using a low dimensional state descriptor (the authors suggest that is due to the repetition of actions or the efficiency of convolutional layers in extracting relevant features from images).

Algorithm 3 DDPG. We use θ to denote weights in primary network and θ' to denote weights in the target network.

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ.
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q, \theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer R
for episode = 1, M **do**
    Initialize a random process $\mathcal{N}$ for action exploration
    Receive initial observation state s_1
    for t = 1, T **do**
        Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
        Execute action a_t and observe reward r_t and observe new state s_{t+1}
        Store transition (s_t, a_t, r_t, s_{t+1}) in R
        Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
        Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
        Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
        Update the actor policy using the sampled policy gradient:

$$\bigtriangledown_{\theta^\mu} J \approx \sum_i \bigtriangledown_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \bigtriangledown_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}$$

        Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$

$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

5.3.1 Actor-Critic Model Architecture

DDPG is based on actor-critic architecture (illustrated in Fig.2). In our actor-critic approach to deep Q learning, the critic model estimates Q-function values while the actor model selects the most optimal actions for each state based on those estimates.

This is expressed as the final layer of the actor network which determines acceleration, steering, and brake with fully connected sigmoid, tanh, and sigmoid activated neurons. These bound the respective actions within their domains. In the actor network, both hidden layers are comprised of ReLu activated neurons Fig. 3. In the critic model, the actions are not made visible until the second hidden layer. The first and third hidden layers are ReLu activated, while the second merging layer computes a point-wise sum of a linear activation computed over the first hidden layer and a linear activation computed over the action inputs 4.

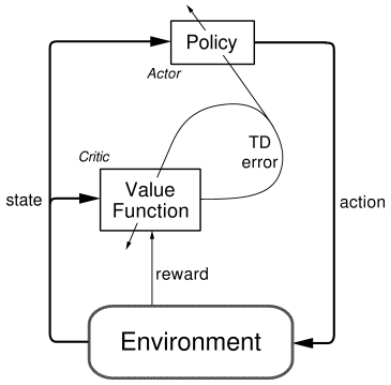

Figure 2: Actor-Critic Model Architecture from Sutton's Book on RL

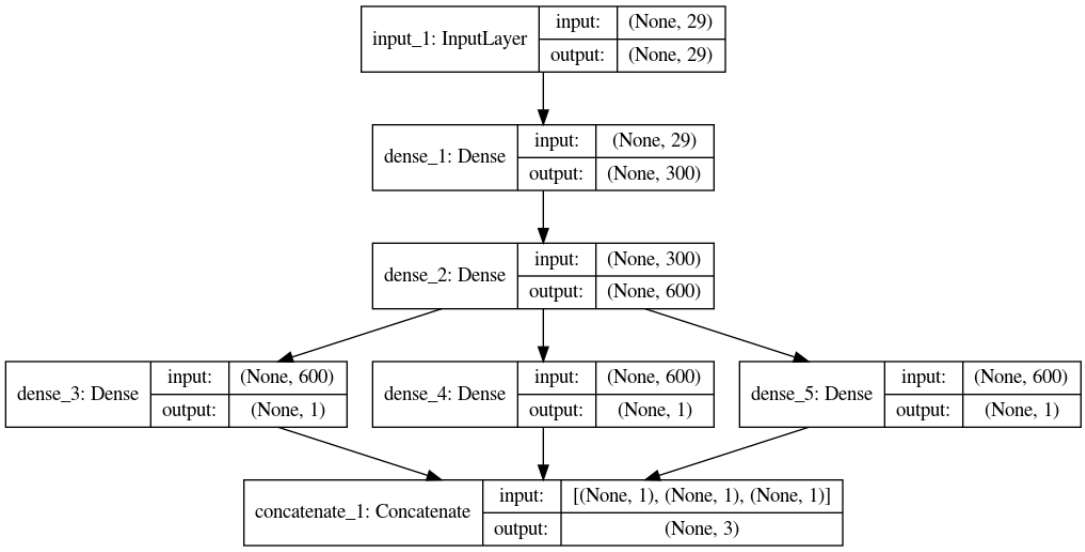

Figure 3: Actor Model Architecture with two ReLu activated hidden layers

5.3.2 Replay Buffer

To diminish the effect of highly correlated training data, we train the neural networks with a large replay buffer D that accumulates 100,000 samples. During training, we update our parameters using samples of experience $(s, a, r, s') \approx U(D)$, drawn uniformly at random.

$$L = E_{s,a,r,s'} \approx U(D)\big[(r + \gamma \max_{a'} Q(s', a') - Q(s, a))^2\big]$$

Using nonlinear functions to approximate the Q Function leads to instability or divergence. Here we exploit experience replay to increase efficiency and mitigate instability by smoothing over changes in the data distribution.

5.3.3 Exploration Algorithm : Ornstein-Uhlenbeck Random Process

To encourage sensible exploration, we use an Ornstein-Uhlenbeck process [9][5] to simulate Brownian-motion about the car with respect to its momentum. This process avoids pathologies of other exploration algorithms that frequently cause the car to brake and lose momentum.

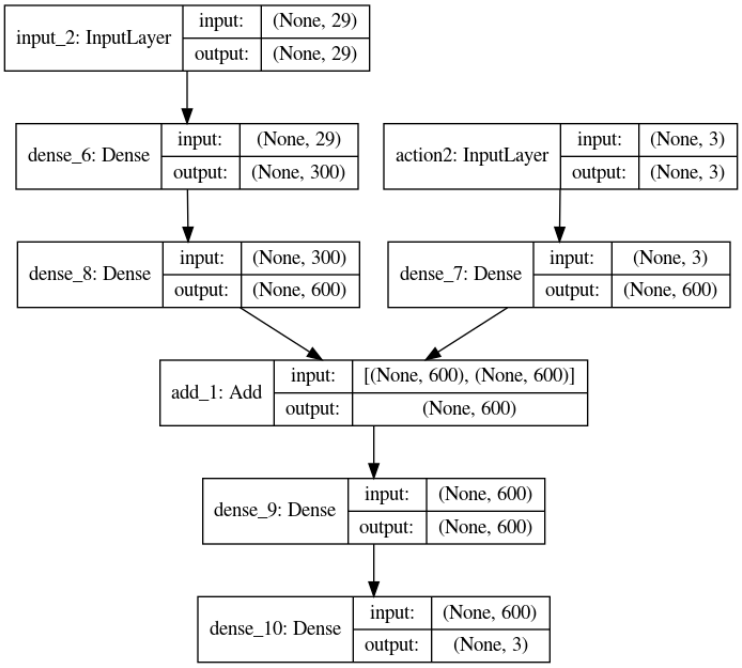

Figure 4: Critic Model Architecture with ReLu and linear activated hidden layers

6 Reward System

As usual with reinforcement learning problems, it is crucial to define a proper reward function so as to achieve proper results. Here we want the car to go as fast as possible, as far as possible all the while staying inside the track. Thus the idea of the authors was to consider the projection of the velocity of the car onto the track axis [9], as shown in figure 5: $R_t = V_x cos(\theta_t)$.

However, they state that the training is not very stable so, as suggested in the implementation we took [8], a more efficient reward is to keep maximising this speed but also minimizing the transverse velocity and tendency of the AI to drive off the center of the track: $R_t = V_x cos(\theta_t) - V_x sin(\theta_t) - V_x |trackPos|$. The problem with this reward is that the model has difficulties to learn how to brake because it diminishes it (hence it will tend to accelerate a lot while it's not outside of the track) which is tackled by imposing a 10% brake during exploration.

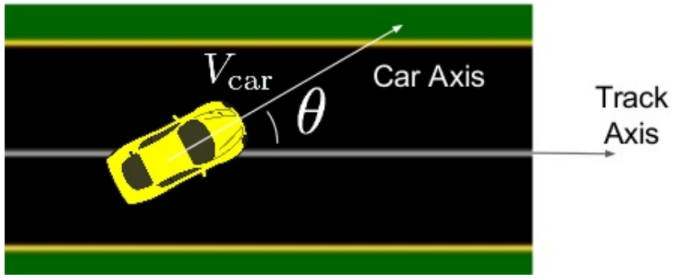

Figure 5: Velocity and its projection [8]

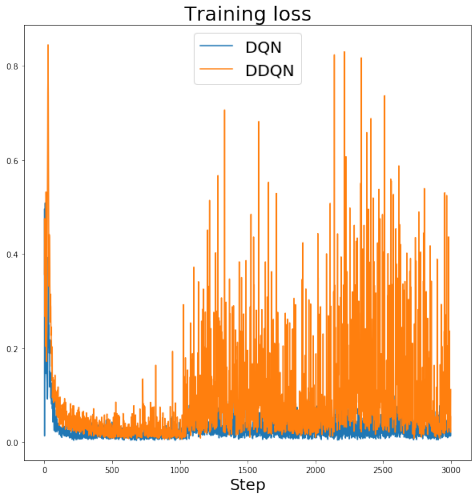

Figure 6: Training loss.

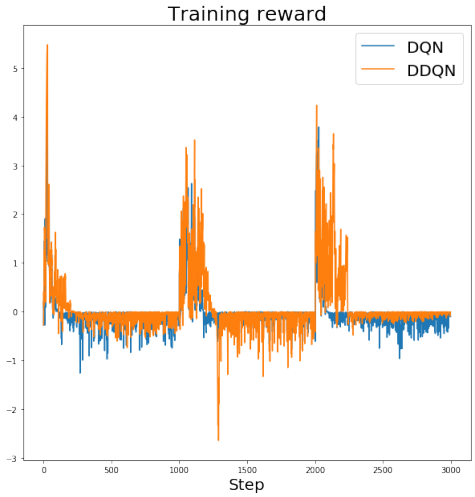

Figure 7: Training reward.

7 Evaluation and Results

7.1 DQN and Double DQN Results

For DQN and double DQN, we discretize the steering action into 30 levels (from -1 to 1), the acceleration action into 15 levels (from 0 to 1), the brake action into 15 levels (from 0 to 1). We train both model with Aalborg track (Fig. 9). The maximum step per each epsiode is set to 1000. Since we don't possess enough computation power, we just train DQN and double DQN for only 3 episodes. The training loss and reward received from each training step is shown in figure 6 and figure 7. The training loss roughly converges but still fluctuates for both models, and the reward obtained during the training phase seems similar but we can observe double DQN is able to obtain slightly better rewards.

We tested both models with the same track without any exploration. The racing car always starts from a straight track. In figure 8, we can observe that DQN try to take a brave action at around 200^{th} step in order to get a better reward; however, this move puts the car in a bad situation and cause it to rush into the side walk and get stuck. On the other hand, double DQN is able to go straight forwards without any difficulty. We can also observe the over estimation problem of DQN. We plotted the predicted maximum state-action value for those three actions at each step and found that DQN always predict the reward higher than double DQN even the car is stuck. From these tests, we can conclude that double DQN is a better model for Torcs.

Although discretization is a intuitive method, we still face some problems. If we don't discretize the action into enough levels, we could not control the car finely. On the other hand, if we discretize the action into too many levels, there would be too many combinations to explore. This makes training more difficult and less effective.

7.2 DDPG Results

Using the Tensorflow[2] and Keras[4] deep learning frameworks, we trained our Actor and Critic Networks. The implementation and hyperparameter choices were adapted from [8]. A replay buffer size of 100000 state-action pairs was chosen, along with a discount factor γ of 0.99. The Adam [7] Optimizer was applied with batch size of 32 and learning rate of 0.0001 and 0.001 for the actor and critic networks respectively. Target networks were trained with gradually updated parameters, as described in [12], which stabilizes the convergence. Training was performed on Aalborg track (Fig. 9) for 500 episodes with each episode having 1000 steps (i.e 500000 iterations) on NVIDIA GeForce GTX 1060 GPU. Fig.10 represents reward value with respect to number of iterations during the training phase.

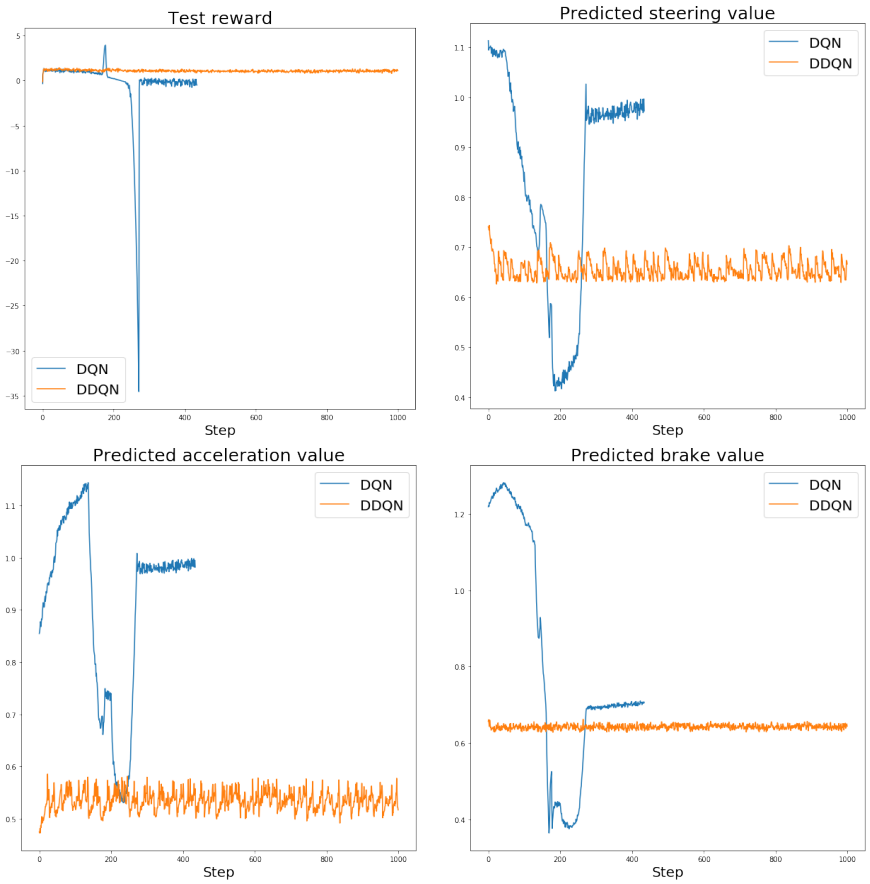

Figure 8: The upper-left figure shows the test reward for both model at each step. The upper-right figure shows the maximum predicted state-action value for steering action. The bottom-left figure shows the maximum predicted state-action value for acceleration action. The bottom-right figure shows the maximum predicted state-action value for brake action.

For validation of our model, we tested it on a simple A Speedway Track (Fig.11) and Challenging Alpine-1 track (Fig.13) for 5 episodes (i.e 5000 iterations). The reward value with respect to number of iterations for Simple A Speedway Track and Challenging Alpine-1 Track is shown in Fig.12 and Fig.14. With this architecture, the resulting agent was capable of navigating a variety of test tracks of highly variable shape in TORCS.

8 Conclusion and Future Work

This research demonstrates a deep Q-learning, double deep Q-learning and deep detewrministic policy gradient techniques that can autonomously drive in TORCS with a suitable reward function, with robustness over diverse environments including the Aalborg, Alpine-1, and A-Speedway TORCS tracks.

In the future, we would like to test a variety of reward functions, exploration policies and also use other challenging tracks to optimize the likelihood of near-optimal and rapid convergence and also dive into transfer learning techniques.

References

[1] Torcs webpage. `http://torcs.sourceforge.net/`.

Figure 9: Aalborg Track

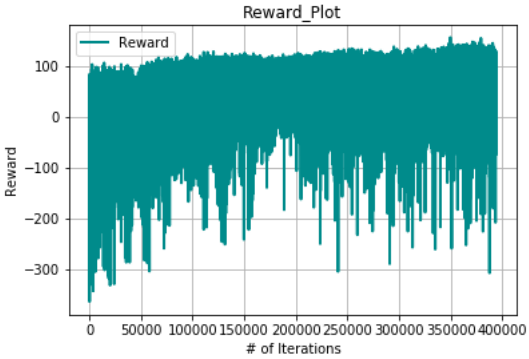

Figure 10: Reward Value per Iteration during Training Process on Aalborg Track

Figure 11: A Speedway Track

Figure 12: Reward Value per Iteration during Validation Process on A Speedway Track

Figure 13: Alpine-1 Track

Figure 14: Reward Value per Iteration during Validation Process on Alpine-1 Track

[2] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[3] Chenyi Chen, Ari Seff, Alain Kornhauser, and Jianxiong Xiao. Deepdriving: Learning affordance for direct perception in autonomous driving. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2722–2730, 2015.

[4] François Chollet et al. Keras. `https://keras.io`, 2015.

[5] Joseph L Doob. The brownian movement and stochastic equations. *Annals of Mathematics*, pages 351–369, 1942.

[6] Hado V Hasselt. Double q-learning. In *Advances in Neural Information Processing Systems*, pages 2613–2621, 2010.

[7] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

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

[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. *CoRR*, abs/1509.02971, 2015.

[10] Daniele Loiacono, Luigi Cardamone, and Pier Luca Lanzi. Simulated car racing championship: Competition software manual. *CoRR*, abs/1304.1672, 2013.

[11] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[12] 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):529, 2015.

[13] ugo-nama kun. Gym-torcs repository github. `https://github.com/ugo-nama-kun/gym_torcs`.

[14] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.