Deep Reinforcement Learning on Self-Driving Cars via the TORCS Simulator

Lou Zhang
Virginia Tech
Arlington, Virginia
lzhang02@vt.edu

Sergio Sainz-Palacios
Virginia Tech
Oakton, Virginia
ssainz@vt.edu

ABSTRACT

Self-driving cars are an important application of deep learning that has the potential to have real economic consequences. In this study, we will compare and contrast reinforcement learning techniques for the self-driving car application in TORCS (The Online Car Racing Simulator), as well as introduce new modules that improve performance and extend current architectures to be platform-agnostic.

CCS CONCEPTS

• **Theory of computation → Reinforcement learning**; • **Computer systems organization → Neural networks**; • **Software and its engineering →** *Virtual worlds training simulations*;

KEYWORDS

Deep learning, video games, self-driving cars, reinforcement learning

1 INTRODUCTION AND PROBLEM STATEMENT

TORCS is an open-source racing game geared towards self-driving enthusiasts and AI developers. There is an annual tournament-style competition that attracts submissions from many academicians and industry experts. We were motivated to select this platform and competition largely because of the open-source nature of the TORCS platform, which is friendly towards heavy modding, as well as the existing deep reinforcement learning structures created by industry experts, which we based the structure of our project off of. Our project charter consisted of two major goals:

(1) To take these existing structures and tune their parameters and add additional modules so as to achieve faster and more efficient learning
(2) To extend these structures to be platform agnostic, so they can be used in any other racing game regardless of whether we have access to the source code

We achieved the first goal and have set out the structure for the second. We will demonstrate our rationale behind these improvements, show the respective learning curves, and include a few clips of before/after driving performance. We also contribute a general framework, integrating existing modules from diverse domains, to achieve a platform-agnostic deep reinforcement learning agent, which reads in screen pixels and outputs commands to any game with a joystick input.

2 PROJECT OUTLINE

2.1 Platform Evaluation

In planning this project, we performed considerable due diligence into the right platform and architectures to base our project off of. The reason we did not decide to build a self-driving car from scratch is because the engineering required to setup the interface, GUI, etc. would have been prohibitive for the time frame of this project. Thus, the first couple of days were spent evaluating several different racing game platforms and the feasibility of building a deep-learning module on top of them. We debated several games, but ultimately settled on TORCS [5] [13] for the reasons mentioned in the introduction. Especially attractive was the fact that TORCS has an active research community around it, and we thought it was interesting to see one game have so much focused attention on it from multiple universities and institutions.

To understand the reason behind this, we dove deeper into the literature around TORCS and found many well-cited papers both referencing TORCS in deep learning applications (Lillicrap et al) [4], and specifically addressing the art of interacting with TORCS to create these applications (Loiacono et al) [5]. It seemed like TORCS had an early foothold as the leader in open-source AI development in the virtual racing world, the game itself going as far back as 2001. It was not surprising then that we discovered a wealth of code repositories available which set up the basic structure of learning algorithms that can be effectively applied to interact with TORCS. It was also not surprising was that many of these repositories were not well-tuned and required substantial work to optimize. It took us many weeks to get them up and running to see what they were capable of. However, it allowed us to become familiar with the server/client structure of TORCS agents and served as a broad survey of techniques which we could combine to create our own, more effective methodologies. In this way, we stand on the shoulders of giants who came before us – we cannot claim credit for all the code in our project [15] [11] and we cite where they come from, but our contribution is that we meld different techniques and add modules onto them to make them more effective and versatile.

2.2 Existing Codebase

We ultimately selected a structure setup by Dr. Ben Lau from Princeton University to base our improvements off of. Dr. Lau applied what is known as a Deep Deterministic Policy Gradient (DDPG) algorithm [3], which combines Deep Q Learning, Actor Critic methods, and Policy Gradient to achieve a substantial lift on simple Deep Q learning. He connected this to TORCS, showing amateur-level driving after only two hundred episodes. Dr. Lau fortunately left much room for improvement, and structured his code to be easily

built upon by the community (there have been several hundred forks so far). We spent a large portion of our time modifying and adding hyperparameters (such as rewards, termination conditions, etc.) and adding methods on top of his structure (such as batch normalization, reward tracking, etc.) to achieve a more robust driver after less training time.

We also spent a large portion of our time attempting to make Dr. Lau's architecture platform agnostic. Our goal was to be able to apply our augmented DDPG structure to any racing game, not just TORCS. We wanted it to function when would we not have access to the source code of a game or a server/client communication structure. We would need to be able to use essentially the same code structure from game to game. The only way to make this possible was to take raw screen pixels as input, return joystick commands, and gather reward either from image pixels or screen attributes such as lane alignment or speedometers. There exists a sizable amount of collateral to help us perform these functions, and we luckily did not have to set these up from scratch. Rather, we rather spent the majority of our time on these modules fitting the pieces together. A brief survey of modules and our improvements to them is below.

(1) For raw screen pixels, we extracted a module from the "europilot" package in Python [7] which defines an x by x box of screen pixels, and takes a screenshot of this every user-defined milliseconds. These screenshots are both stored in a folder and converted to a numpy array. We hooked these numpy arrays up to structure outlined in "Human-level control through deep reinforcement learning" (Mihn et al) [10], which is a CNN which takes as input the past four images and outputs an action layer. Instead of directly taking the action layer, we take the FC7 layer and use that as the latent representation of the last four images. We were able to substitute the native input (communication from the TORCS server to the agent) with this FC7 layer, representing the pixel data. This effectively allowed us to swap out game states from TORCS specific to game-agnostic, as long as the game outputs an image the user can see (ie. text based games will not work).
(2) For joystick commands, we use the "pyvjoy" package in Python which communicates with any game which takes a joystick input and outputs actions [12]. The package sets bindings for keyboard to joystick commands, and is our substitute for the agent-to-server communication that is native to TORCS.
(3) For rewards, we are still fine-tuning this element but we are aware that tensorflow has an object recognition library. We may use this to help recognize when the car is within 'bounds', or within the right lane lines. Reward could be a function of how long it can stay within bounds and its speed, which can either be extracted from a portion of the screen which contains a speedometer or from how quickly the screen pixels themselves are changing. If the tensorflow implementation of reward proves to be ineffective, we can default to a reward that can purely be extracted from screen pixel change rate.

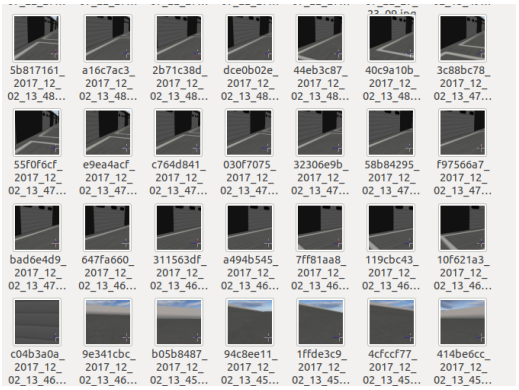

Figure 1: A sample of Europilot's screenshots. These are converted to an array and represent our "states" in the platform-agnostic version

Finally, we should note that we build a Deep Q Network and Deep Deterministic Policy Gradient networks from scratch in tflearn based on the Atari paper [10], and Continuous Control with Deep RL [4] respectively. We compare DQN performance to the DDPG performance and quantify the learning differential.

3 DATASET INFORMATION

The TORCS client offers a rich array of state parameters and action outputs for us to use. The client is essentially an interface to TORCS, which encapsulates the self-driving agent and the necessary communication to the program itself to be functional. Table 1 and 2 in the appendix show the parameters the client can output to any learning agent, and the actions which it can then send to the virtual car. We can use part of these to experiment with different rewards to input the deep reinforcement algorithm. Mainly "damage" (how much damage the car has received) and "trackPos" (the position of the car within the road sidelines) as well as "Track" (19 range sensor indicating distance to the track).

We extract a selection of these states and feed them through our agent, a self-driving robot. The agent ingests these states into a DDPG (Deep Deterministic Policy Gradient) framework, which as previously mentioned consists of a Deep Q network, an Actor Critic network, and a Policy Gradient.

Besides the environment state, we also extract the grayscale image of size 128 by 128 pixels from the TORCS environment. This results in a 16,384 array as input to the convolutional network explained in the section: Convolutional Neural Network (4.4.2).

4 ARCHITECTURE

The architecture of our project uses Dr. Ben Lau's codebase as our backbone. We build improvements on top of his code to achieve faster learning and game-flexibility. Our biggest add-on is the ability to effectively take raw-screen pixels as our 'states' and output joystick commands as our 'actions'.

4.1 Baseline Architecture

Dr. Lau's architecture [3] can be interpreted into three major parts, in which the first two parts draw from work done by Chris X Edwards and Naoto Yoshida [14]. In the interest of focusing more on the improvements we made, we will only briefly go over their work, just enough to obtain necessary background information to understand the whole pipeline.

- Chris X Edwards [1] is responsible for the module that allows communication between the TORCS server (game) and a robot agent. We do not modify this script as it merely provides the technical interface from robot-to-server. His script initializes the robot, which is a wrapper for any algorithms, attributes, and feedback loops that the user wants to put inside of it. The module is a Python script which contains
  (1) The Client class (robot), which contains all necessary attributes for the robot to function as well as a connection to the UDP socket the TORCS server uses to communicate
  (2) The ServerState class, which contains the attributes of the current game state, like car position, distance from start, opponent positions, etc.
  (3) The DriverAction class, which is used to communicate robot actions, like steering, braking, etc., back to the game itself
- Naoto Yoshida is responsible for the module (aptly called gym torcs) which creates a reinforcement learning architecture around TORCS, loosely inspired by the gym (OpenAI) package. It contains all the basic functions for allowing basic RL capabilities, such as rewarding the agent, penalizing the agent, sensing states (i.e. current speed, position, image pixels, etc.), and dynamic updating. We should note that this is the first script which we modify to both improve our learning efficiency and allow for flexibility across platforms.
- Finally, Dr. Lau's work is built on top of these two other modules, and his primary contribution was to build a DDPG algorithm around the gym torcs package. He adds three major modules: the Deep Q Learning module, the Actor Critic network, and the ability to learn via Policy Gradients. These are three scripts which work in tandem according to the DDPG algorithm, which is covered in a deep dive below.

4.2 Deep Deterministic Policy Gradient Networks

Dr. Lau's designed the following neural networks for use in his DDPG experiment and we reuse it for our project as well. It is actually two networks, one for the policy network (actor) and another one for the Q-Value network (critic).

- Policy network (Actor). This takes as input 29 simulation state variables (focus, speedX, speedY, speedZ, damage, opponents, rpm, track, trackPos, trackDist, wheelSpinVel, etc. see appendix for definitions). Then, there are two layers, the first one with 300 neurons, and second one with 600. Both layers have the ReLU activation function. Finally there are three outputs, one for each of the action outputs: the steering variable has a range of -1, 1 and therefore, the activation function used is tanh. The other two variables: brake and acceleration have range of 0 to 1, and therefore their activation function is sigmoid.

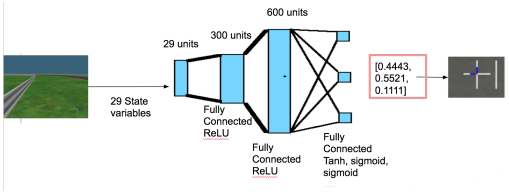

Figure 2: Policy network, actor network

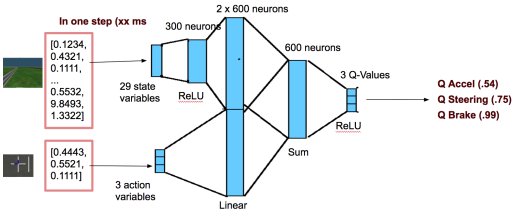

Figure 3: Q-Value network, critic network

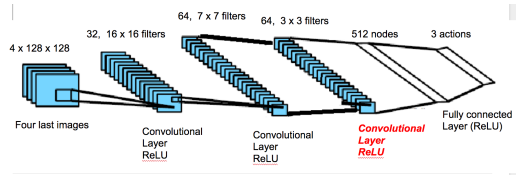

Figure 4: Convolutional Neural Network (in red the new convolution layer added), the weights are also adapted to the new image size

- Q-Value network (Critic). This takes two parts, the simulation state variables and the actions chosen by the agent with those state variables. The state variables are passed to a fully connected layer of 300 units and then to a linear layer of 600 units. Meanwhile the action input variables are passed to a 600 units linear layer. The two 600 units layers are summed up and ReLU activation function applied. Finally another linear fully connected layer is applied containing 3 nodes, these three nodes are the output units representing the Q-Values of each action.

4.3 Improvements and Hyperparameter Tuning

We contribute a number of improvements to the original architecture, adding in the methodological improvements of recent years and bringing Dr. Lau's architecture up to speed with current advancements.

(1) We add **batch normalization** to both the actor and the critic networks. Batch normalization has in theory shown to

improve the efficiency of learning and flexibility of parameters tolerated by normalizing each batch by its mean and variance [9]. We wanted to test empirically if this improvement applied to TORCS as well. Chief among our challenges was the determination of where to apply batch norm within all the layers of the Actor Critic network. There exists some literature on this though nothing universal for all domains. We ultimately decided to place the batch normalization layer right before the final layer in both the actor and critic networks. We reasoned that this would allow the input to the action layer to be standardized within a range before being fed to the final layer which uses tanh to squish values between -1 and 1. This is important because wildly varying input values would cause tanh to output values that were vastly far apart on this scale. In the driving domain, it is rare to see extremely sudden shifts in steering, acceleration, and brake even if the course is difficult to navigate. Thus we want to discourage these tendencies. A similar logic holds for the critic network – we do not believe the Q function will experience rapid shifts based on input from the actor network. Batch normalization should temper these variances. Empirically, we show that there is indeed an improvement from utilizing batch normalization (Amendment 1).

(2) We tune a number of **hyperparameters**– including termination conditions, the reward function, and afterwards the game parameters.
- We add in the ability to terminate the episode after a certain number of unproductive states. We reasoned that there are many circumstances where the car can get stuck in a local minimum in the reward space (i.e. stuck against a wall) until the max steps are reached and the episode terminates. This is because driving is inherently a domain where the "right" set of actions is very sparse. The number of correct actions is dwarfed by the number of incorrect actions, and as errors accumulate the probability of executing the correct sequence of actions to recover becomes extremely small. We evidenced this just by watching the baseline algorithm performance – we saw many hours where the car simply ran up against a wall, with the robot not realizing that the correct sequence of events to "get back on track" was to reverse and turn.
- Initially the reward function is defined as
  $\mathbf{reward} = (\mathbf{speed} * \mathbf{sin}(\mathbf{angle})) - (|\mathbf{speed} * \mathbf{cos}(\mathbf{angle})|) - (\mathbf{speed} * |\mathbf{track\ position}|)$
  The angle is between the car's direction and the road axis. Track position refers to the distance between the car and the road axis. This reward function encourages the car to drive parallel to the road axis and as fast as possible.
  We tried to tune the reward function to be simply speed vs. speed and track position, but did not realize an increase in learning with any other combination of improvements. We initially reasoned that the car would learn novel routes around the track, cutting corners and revealing navigation strategies that were not immediately obvious to humans. We thought it would at first bounce around the walls quite a bit, but eventually learn that hitting walls was not productive towards speed. We did see the presence of less wall-hits over time, but unfortunately, a gain on performance was never realized even after hundreds of episodes. This may be because speed itself is insufficiently complex as a reward, or because we just needed to train longer (there seemed to be "jumps" in performance rather than a smooth increase with a speed-only reward).
- Finally, we tuned the game parameters such as car speed, tracks, presence of other cars, etc. to increase the robustness and flexibility of our agent.

Figure 5: The agent struggles to get out from being stuck against the wall. This is mitigated by adding a termination counter when the car hits the wall and has slow speed. Once a certain number of termination counters are exceeded, the episode terminates allowing more productive state-action pairs to be learned

4.4 Screen Pixel Module

(1) Dr. Lau's architecture does not support image-pixel states very well. It was limited to very small images (limit to 64 by 64), and did not process the raw pixels at all (the state was a 3,000 element array). The performance was unsurprisingly poor when using the default settings, showing no episodes where the average reward per step exceeded 0. This means the agent was more often than not either hitting the wall or going out of bounds.

(2) **Convolutional Neural Network** In an effort to make the learning process agnostic of the simulator we started an endeavor of taking as input the simulator's images and as output the actions the agent should take. We used same convolutional neural network in both the DDPG algorithm and Deep Q-Learning algorithm. This network is based off the Playing Atari paper (Mnih et. al.) [10], but we change the network filter size in addition to adding a new convolutional layer to the end of the network to reduce the image size. The main reason for this change is because the image we obtain from simulator is 128x128 instead of 28x28 as in Atari's network. The input to the convolutional network is a stack of the last four images as seen by the agent; the input size will be 4 times 128 times 128. The first layer is a convolution of 32 filters of size 16 by 16 with stride four. The second layer is a

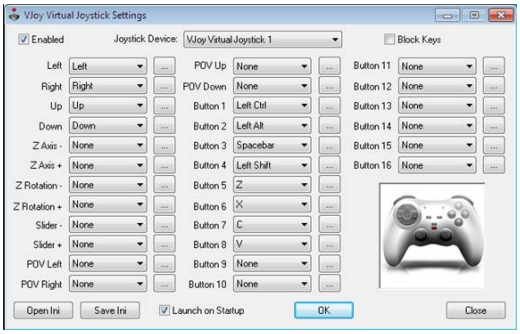

Figure 6: pyvjoy key bindings. A program (in this case our ddpg.py script) can push actions to the controller which are then communicated to the game

convolution of 64 filters of size 7 by 7 with stride two. Third layer is a convolution of 64 filters of size 3 by 3 with stride one. For all convolutional layers the activation function is ReLU. The next layer is a fully connected layer of size 512 wih ReLU activation function. This 512 units fully connected layer is used to predict all the action's Q-Values in the case of Deep Q-Learning, or used as input to the Actor and Critic networks in the case of Deep Deterministic Policy Gradient.

4.5 Future Work: Building to be Platform Agnostic

In our effort to become platform-agnostic, we tap into two other resources – the screen capture capability from the europilot package, and the pyvjoy package which enables communication with any game that allows for joystick input. We ran into several challenges with assembling these all together

(1) Europilot's array feed mechanism initially resulted in many timeouts with the TORCS client [7]. We realized that we needed to modify the start condition of the TORCS server to except communication every 100 ms vs. every 10 ms by default. This is because the Europilot module captures screenshots every user-defined ms. The default settings were not aligned, and when we tried to align Europilot to TORCS' conditions, taking screenshots every 10 ms, we found it was not feasible with our hardware. Thus, our solution was to trade bandwidth for compatibility, modifying TORCS to except communication 10x less frequently than by default.
(2) pyvjoy is one of many modules which can issue keyboard-/joystick commands [12]. Our next steps include feeding the action array (output by the actor network) into pyvjoy to associate each array element to a key binding. For example, the current action array takes the form [steering, acceleration, brake], which is sent back to the TORCS server every few milliseconds. These can be mapped to steering (values mapped to unit circle, and then to the Z Rotation button), acceleration mapped to the X button, and brake mapped to Z button.

5 ALGORITHMS INVESTIGATED

We can define Reinforcement Learning (RL) as learning a desired agent behavior through interaction with the environment. RL agents will perform a series of trial and error actions on an environment and analyze consequences of its actions with respect to a desired outcome. The desired outcome is defined numerically through a reward function. After enough trial and error, the agent will find the optimal series of actions that maximize the reward. That is, the RL agent automatically learns to take actions that optimize a reward function over time. The series of steps that optimize the reward function is formally known as an optimal policy. The RL agent uses diverse algorithms, like Q-Learning, adaptive actor-critics or temporal difference learning to learn the optimal policy.

In our work we evaluated two algorithms – Deep Q-Learning (DQN) and Deep Deterministic Policy Gradient (DDPG).

5.1 Deep Q-Learning

Vanilla Q-Learning is trying to learn a function Q that defines the expected accumulated reward of taking action a, given agent is in state s: Q(s, a). Once the Q function converges, we can then find the optimal policy by taking at each state the action that returns the highest Q function value. Traditionally, Q-Learning uses exhaustive search among all actions taken from all states to converge to the best policy [8]. But this is not convenient if the number of states is very large. Also, in traditional Q-Learning, there is no way to know that two states are very similar to each other and consequently the Q function should return similar expected accumulated reward output.

Deep Q-Learning is introduced here to allow approximation of the Q function by using an artificial neural network to learn the Q function. By using a neural network to define the Q function we can take advantage of the convolutional neural network architectures that learn to identify patterns within environment state to make efficient Q function evaluations. Also, by using the neural network as a Q function, not all the state-action pairs' Q-value need to be learned, only the ones that the neural network is exposed to. And even if a new state never seen before appears, the neural network may return a good approximation because the neural network learns to identify patterns within the state.

When applying Deep Q-Learning algorithm to the TORCS problem, we faced the following challenge: DQN is good to predict the Q-values of binary actions, but it does not easily calculate the Q-Values for continuous actions. The reason for it is because the network outputs all the possible action's Q-Values. For continuous actions this becomes unfeasible. TORCS has continuous actions, and hence we could not easily predict the Q-Values for its actions.

The workaround for this challenge was to discretize the three continuous actions into combinations of the three variables. We discretize the steering wheel action into 9 parameters, the acceleration and brake actions are each discretized into 7 parameters. That gives to a total of 13 times 9 times 9 actions, 1,053 actions. Then the DQN network takes as input four 128 by 128 images and outputs Q-Values for 1,053 actions. The agent converts the chosen action into equivalent three values of each of the action parameters (steering, acceleration and brake).

ALGORITHM 1: Deep Q-Learning

Initialize replay memory D with size N;
Initialize action-value function Q with random weights ;
for *episode* = 1, M **do**
  Initialize sequences $s_1 = \{x_1\}$;
  for $t = 1, T$ **do**
    With probability of ϵ select random action a_t ;
    Otherwise select a_t as $\max_a Q(s_t, a; \theta)$;
    Execute a_t in emulator get reward r_t and image x_{t+1} ;
    Set $s_{t+1} = s_t, a_t, x_{t+1}$;
    Store transition (s_t, a_t, r_t, s_{t+1}) in D ;
    Sample random minibatch transitions (s_j, a_j, r_j, s_{j+1}) from D;
    Set $y_j = r_j$ for terminal s_{j+1} for terminal s_{j+1} ;
    Set $y_j = r_j + \gamma \max_a Q(s_{j+1}, a; \theta)$ for non terminal s_{j+1};
    Perform gradient descent on $(y_j - Q(s_j, a_j; \theta))^2$;
  end
end

5.2 Deep Deterministic Policy Gradient

The second algorithm we tested is: The Deep Deterministic Policy Gradient (DDPG), described in this reference [4]. It is one of the most recent algorithms released in the autonomous driving domain. It is a combination of the Deep Q, Actor-Critic, and Deterministic Policy-Gradient algorithms. After looking at several competitors in the most recent TORCS challenge, we found that DDPG was a commonly used algorithm across many agents [3][6].

The intuition behind DDPG is that there are two networks, one in charge of deciding the action (policy network) and another one in charge of calculating the Q-Value of the action and apply gradient descent on both the networks. Because based on Q-Value network loss function gradient the weights of both its own Q-Value network as well as the policy network are updated it is also called critic network. The loss function used is mean squared error with the expected Q-values. The expected Q-values are calculated by taking a random sample from previous states and updating Q-Value function by adding the reward information obtained from the environment. Formally the sample is defined as (state, action, reward, next state), and the expected Q-value is Q(state, action) equal to reward + Q(next state, μ (next state)). Where $\mu(.)$ is the policy network.

DDPG uses some of the innovations included in [10], for example: replay buffer memory to achieve better estimation of independent and identically distributed samples when applying stochastic gradient descent.

It also uses two sets of networks. One set is to calculate the actual agent's actions, while the other one is used to calculate the expected Q-Values; we call the latter set the target networks ($\theta_{Q'}$). The target networks are marginally updated every learning step with the weights of the actual networks: $\theta_{Q'} \leftarrow \gamma\theta^Q + (1-\gamma)\theta^{Q'}$. Here γ is a small value (.001 in our case), and it specifies how fast the target networks are updated with the actual network weights. The target critic network weights and target policy network weights

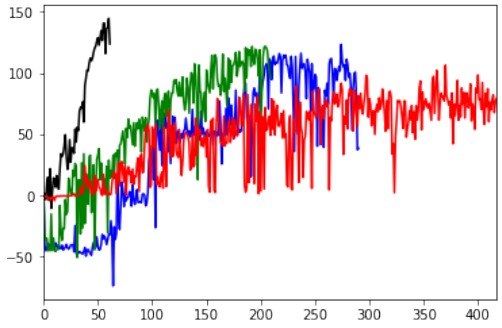

Figure 7: A sample of test results, with average reward per step. Episodes are on the X axis, reward on the Y. Red is our baseline, with the others as various improvements outlined in Amendment 1

are $\theta_{Q'}, \theta_{\mu'}$ respectively. Meanwhile the critic network weights and policy network weights are θ^Q, θ^μ respectively.

ALGORITHM 2: Deep Deterministic Policy Gradient

Initialize critic network $Q(s, a; \theta^Q)$ N;
Initialize actor $\mu(s|\theta^\mu)$ with weights θ^Q θ^μ;
Initialize target network $Q'\mu'$ with weights $\theta^{Q'} \leftarrow \theta^Q$
$\theta^{\mu'} \leftarrow \theta^\mu$;
Initialize replay buffer R **for** *episode* = 1, M **do**
  Initialize random process N for action exploration ;
  Receive initial exploration state s_1 ;
  for $t = 1, T$ **do**
    Select a_t as $\mu(s_t|\theta^\mu) + N_t$ according with current policy and exploration noise ;
    Execute a_t in emulator get reward r_t and state s_{t+1} ;
    Store transition (s_t, a_t, r_t, s_{t+1}) in R ;
    Sample random minibatch transitions (s_j, a_j, r_j, s_{j+1}) from R;
    Set $y_j = r_j + \gamma Q'(s_{j+1}, \mu'(s_{j+1}|\theta_{\mu'})|\theta_{Q'})$;
    Update critic by minimizing $L = \frac{1}{N}(\sum_j (y_j - Q(s_j, a_j|\theta^Q))^2$;
    Update the actor policy using the sampled policy gradient: ;
    $\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_j \nabla_a Q(s, a|\theta^Q)|_{s=s_j, a=\mu(s_j)} \nabla_{\theta^\mu}(s|\theta^\mu)|_{s_j}$;
    Update target networks ;
    $\theta_{Q'} \leftarrow \gamma\theta^Q + (1-\gamma)\theta^{Q'}$;
    $\theta_{\mu'} \leftarrow \gamma\theta^\mu + (1-\gamma)\theta^{\mu'}$;
  end
end

6 RESULTS

Complete results can be found in Amendment 1: Test Results. In order to compare different algorithms and different models we define the following metrics:

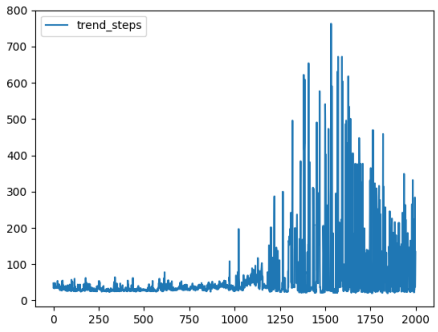

Figure 8: Deep Q-Learning, steps per episode. Discretize continuous variables into 1053 combinations of action variables. Steering split in 13 values, acceleration into 9 and brake into 9 values.

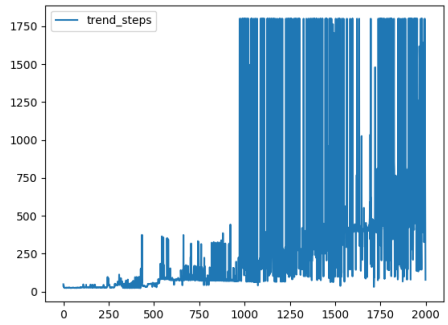

Figure 9: Deep Deterministic Policy Gradient, steps per episode.

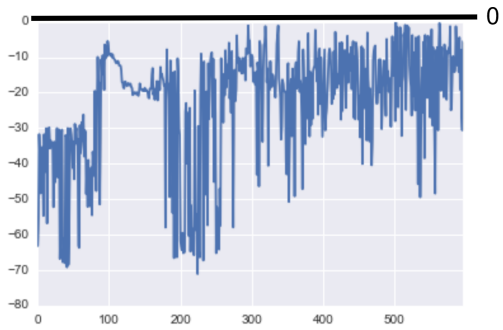

Figure 10: Normalized rewards. DDPG + image: Input the four 128 by 128 images as a vector into the DDPG networks (no CNN involved to process image)

(1) Rewards per episode: how many rewards the robot takes each episode. This is a measure of how efficient the algorithm is.
(2) Steps per episode: how long each race takes before termination. This is a measure of how soon the robot learns the rules of the game.
(3) Normalized rewards per episode: on average, how many rewards per step is achieved within each episode. This is a normalized measure of the first metric which we found to be a better yardstick for comparing different methods.

6.1 Deep Q-Learning and Deep Deterministic Policy Gradient Comparison

As expected, the result is that the Deep Q-Learning algorithm does not performs as well as the Deep Deterministic Policy Gradient algorithm. As mentioned in the Deep Q-Learning section, the DQN is performant with binary actions. Because we discretize the continuous actions of steering, acceleration and brake, we lost a lot of action domain knowledge when defining only a few possible combinations among all possible combinations of these three continuous actions.

We used the metric: Step by episode, the more steps agent achieves in an episode the better learning the agent has achieved (at least learning to avoid ending states). We can analyze the result and comment that while DQN achieves greater steps per episodes starting around 1250th episode, the metric later decreases slightly likely due to the exploration phase decreasing (epsilon) in later phases of the learning.

Meanwhile, DDPG algorithm starts learning as early as episode 600 and it actually reaches the maximum limit of steps by episode 1000th episode. After this result we use DDPG for further experimentation.

6.2 Deep Deterministic Policy Gradient with Image as Input

In an effort to make the learning algorithm agnostic of the driving car simulator, we need to train a model that can learn from image instead of simulator dependent variables. The following are three iterative experiments we did to test the reinforcement learning from image, in particular using DDPG (instead of DQN).

(1) Image is used as input DDPG network.
The first attempt to integrate the image as input to the DDPG networks was to plainly insert the 128 times 128 vector as the input to the DDPG actor and critic network instead of the previous 29 size vector. The results were not positive. The normalized reward is always under 0.
(2) Convolution is applied to image prior inserting into DDPG. Convolution has fixed weights.
The second attempt at integrating the image to the DDPG networks was to pass the 128 times 128 vector through the convolution neural network described in section: Convolutional Neural Network. Notice the weights in this experiments are fixed and randomly initialized to a uniform distribution, where minimum, maximum ranged between $(-\sqrt{\frac{1}{\text{vector size}}}, \sqrt{\frac{1}{\text{vector size}}})$.
In this experiment, we see the normalized reward improves compared with the previous experiment where the image is passed into the DDPG networks without CNN.

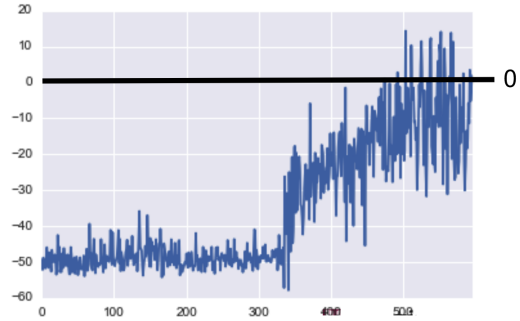

Figure 11: Normalized rewards. DDPG + CNN + Image: Input the four 128 by 128 image as vector into the CNN. Later use the FC7 layer of the CNN to input to the DDPG network. CNN weights are random and not learned.

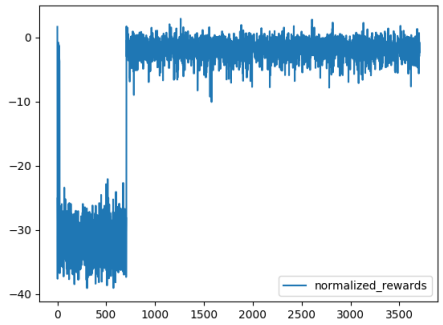

Figure 12: Normalized rewards. DDPG + CNN + Image: Input the four 128 by 128 image as vector into the CNN. Later use the FC7 layer of the CNN to input to the DDPG network. CNN weights are learned.

(3) Convolution network is integrated into the DDPG network. Our third attempt to integrate image into the DDPG network is to use the four consecutive images of 128 x 128 size each as input the convolutional neural network and then use the CNN's FC7 layer as input to the DDPG policy and Q-value networks. We clipped the gradients to -1, 1 as well (as in Atari [10]). In this experiment the weights are randomly initialized from uniform distribution as in the previous experiment. And this time the weights of the CNN are being learned as well together with the DDPG stochastic gradient descent step. The results are that the learning is considerably slower compared with the previous approaches. In order to speed up learning of the model we plan to add more regularization techniques like dropout, and also add batch normalization into the CNN as explained by Ioffe et. al. [2] in the next iteration.

We present the figure with the steps per episodes for this experiment; we can comment that the agent cannot go beyond the 100 steps per episodes in 3500 episodes (while the DDPG from the original inputs (track position, etc.) achieves the maximum allowable steps (2000) in around 1250 episodes).

7 CONCLUSION

TORCS is an excellent platform for developing self-driving agents. Leveraging existing architecture, we were able to setup additional modules which improved on the performance and flexibility of current work. We have contributed to the TORCS and self-driving community in two major ways:

- We have demonstrated marked improvements in learning efficiency after tuning hyperparameters and adding in batch normalization
- We have started the process to make our work platform-agnostic, connecting modules for screenshot-to-state conversion and action-to-joystick communication

Future work includes expanding and testing our architecture in other racing games and hopefully evolving from the virtual to physical domain. Self-driving cars will continue to be a popular space, as the commercial and societal value of autonomous driving is obvious to industry, government, and academia. We hope our work can just be one piece which furthers the development of this technology.

REFERENCES

[1] Chris X Edwards. [n. d.]. Snake Oil. ([n. d.]). Retrieved October 28, 2017 from http://xed.ch/p/snakeoil/

[2] Christian Ioffe, Sergey; Szegedy. [n. d.]. Batch Normalization: Accelerating Deep Network Training by reducing internal covariate shift. ([n. d.]). Retrieved March 2, 2015 from https://arxiv.org/abs/1502.03167

[3] Ben Lau. [n. d.]. Using Keras and Deep Deterministic Policy Gradient to play TORCS. ([n. d.]). Retrieved October 28, 2017 from https://yanpanlau.github.io/2016/10/11/Torcs-Keras.html

[4] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. 2015. Continuous control with deep reinforcement learning. *CoRR* abs/1509.02971 (2015). arXiv:1509.02971 http://arxiv.org/abs/1509.02971

[5] Daniele Loiacono, Luigi Cardamone, and Pier Luca Lanzi. 2013. Simulated Car Racing Championship: Competition Software Manual. *CoRR* abs/1304.1672 (2013). arXiv:1304.1672 http://arxiv.org/abs/1304.1672

[6] Xiyao Ma. [n. d.]. TORCS car github. ([n. d.]). Retrieved October 28, 2017 from https://github.com/Shawn617/Torcs_car

[7] MarsAuto. [n. d.]. sEuropilot repository. ([n. d.]). https://github.com/marsauto/europilot

[8] Francisco S. Melo. [n. d.]. Convergence of Q-learning:a simple proof. ([n. d.]). Retrieved October 28, 2017 from http://users.isr.ist.utl.pt/~mtjspaan/readingGroup/ProofQlearning.pdf

[9] Dmytro Mishkin. [n. d.]. Batch Normalization. ([n. d.]). Retrieved November 28th, 2017 from https://github.com/ducha-aiki/caffenet-benchmark/blob/master/batchnorm.md

[10] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. 2013. Playing Atari with Deep Reinforcement Learning. *CoRR* abs/1312.5602 (2013). http://arxiv.org/abs/1312.5602

[11] Sergio Sainz. [n. d.]. DDPG. ([n. d.]). https://github.com/ssainz/ddpg

[12] Matt Saunders. [n. d.]. PyJoy. ([n. d.]). https://github.com/tidzo

[13] Bernhard Wymann. [n. d.]. The Open Racing Car Simulator. ([n. d.]). Retrieved October 28, 2017 from http://torcs.sourceforge.net/

[14] Naoto Yoshida. [n. d.]. Gym TORCS. ([n. d.]). Retrieved October 28, 2017 from https://github.com/ugo-nama-kun/gym_torcs

[15] Lou Zhang. [n. d.]. DDPG. ([n. d.]). https://github.com/apclypsr/DDPG-Keras-Torcs

Table 1: TORCS simulation environment state parameters. Bold parameters used as input.

Parameter	Range	Meaning
angle	$[-\pi, \pi]$	**Angle between the car direction and the direction of the track axis**
curLapTime	$[0, +\infty)$ (s)	**Time elapsed during current lap.**
damage	$[0, +\infty)$ (point)	**Current damage of the car**
distFromStart	$[0, +\infty)$ (m)	**Distance of the car from the start line along the track line.**
distRaced	$[0, +\infty)$ (m)	**Distance covered by the car from the beginning of the race**
focus	$[0, 200]$ (m)	**Vector of 5 range finder sensors: each sensor returns distance between track edge and the car**
fuel	$[0, +\infty)$ (l)	**Current fuel level.**
gear	$\{-1, 0, 1, 6\}$	**Current gear: -1 is reverse, 0 is neutral and the gear from 1 to 6.**
lastLapTime	$[0, +\infty)$ (s)	**Time to complete the last lap**
opponents	$[0, 200]$ (m)	**Vector of 36 opponent sensors: returns the distance of the closest opponent in the covered area**
racePos	$\{-1, 0, 1, N\}$	**Position in the race with respect to other cars.**
rpm	$[0, +\infty)$ (rpm)	**Number of rotation per minute of the car engine.**
speedX	$(-\infty, +\infty)$ (km/h)	**Speed of the car along the longitudinal axis of the car.**
speedY	$(-\infty, +\infty)$ (km/h)	**Speed of the car along the transverse axis of the car.**
speedZ	$(-\infty, +\infty)$ (km/h)	**Speed of the car along the Z axis of the car.**
track	$[0, 200]$ (m)	**Vector of 19 range finder sensors**
trackPos	$(-\infty, +\infty)$	**Distance between the car and the track axis.**
image	$(64 * 64, +\infty)$	**Image vector of size 128 by 128, greyscale**

Table 2: TORCS simulation action parameters. Bold parameters are predicted.

Parameter	Range	Meaning
accel	$[0, 1]$	**Virtual gas pedal (0 means no gas, 1 full gas)**
brake	$[0, 1]$	**Virtual brake pedal (0 means no brake, 1 full brake).**
clutch	$[0, 1]$	**Virtual clutch pedal (0 means no clutch, 1 full clutch).**
gear	$[-1, 0, 1..., 6]$	**Gear value.**
steering	$[-1, 1]$	**Steering value: -1 and +1 means respectively full right and left**
focus	$[-90, 90]$	**Focus direction (see the focus sensors in appendix) in degrees.**
meta	$[0, 1]$	**1: Restart simulation, 0 is no action**