Reinforcement Learning for Robot Path-Following

Hadi Hafeez
University of Toronto
Mississauga
UtorID: hafeezh1
`hadi.hafeez@mail.utoronto.ca`

Mahan Mohammadi
University Of Toronto
St. George
UtorID: moha1855
`mahan.mohammadi@mail.utoronto.ca`

Hamza Faisal
University of Toronto
Mississauga
UtorID: faisal17
`h.faisal@mail.utoronto.ca`

Santiago Garrido
University Of Toronto
Missisauga
UtorID: garrid10
`santiago.garrido@mail.utoronto.ca`

Abstract: Proportional-integral-derivative (PID) feedback control is one of the most commonly used controllers for enabling a robot, which obeys differential drive dynamics, to follow a path and avoid obstacles. However, throughout recent years, Reinforcement Learning (RL) has been used significantly in various areas of robotics including robot path-following and obstacle avoidance. In this research project, we will use Deep Reinforcement Learning (DRL) with actor-critic structures to allow a robot to follow a path, defined as the middle of the track in a simulated car racing track environment. Our learning algorithm implements the Twin-Delayed Deep Deterministic Policy Gradient (TD3) algorithm which we utilize with certain modifications to be more specific to our particular problem. In our findings, we were able to reach convergence to a decent steering policy fairly quickly. While initially the race car oscillated along the path, we managed to solve this by shaping the reward function which allowed the race car to follow and stay close to the center of the track. The model was also trained for steering at various speeds successfully. Our future prospects include giving the car control for acceleration and brake along with steering as a more challenging problem.

Keywords: Proportional-integral-derivative feedback (PID) control, Path-Following, Reinforcement Learning (RL), Deep Reinforcement Learning (DRL), Twin-Delayed Deep Deterministic Policy Gradient (TD3), Simulated Car Racing Track Environment

1 Introduction

PID is a control loop feedback mechanism used to achieve many tasks in the field of robotics ranging from path-following tasks to altitude control of aerial robots. Although PID control is still popular, it requires manual tuning of parameters (Proportional Gain, Derivative Gain, Integral Gain) by experts. This is why over recent years, Reinforcement Learning (RL) has been increasingly used in robotics tasks like control design and path planning. RL is about the study of agents and how they autonomously learn via trial and error by performing actions within an environment to maximize rewards and achieve the best outcomes. In comparison to PID control or to the more automatic self-tuning PID controllers, RL has better accuracy, smoothness, adaptability, and robustness and can learn optimal control in real-time within complex uncertain environments [1,4].

In addition, Deep Reinforcement Learning (DRL) combines reinforcement learning and deep learning and allows efficiently and effectively an autonomous agent to learn optimal behaviors to solve complex problems and adapt to uncertain environments. DRL algorithms with actor-critic structures like Deep Deterministic Policy Gradient (DDPG) and Twin-Delayed Deep Deterministic Policy Gradient (TD3) have been increasingly used for finding optimal control policies within different control design tasks. DDPG is a deterministic policy gradient operating in continuous action spaces [2]. However, considering that DDPG suffers in complex environments by overfitting the current policies [6], TD3 was proposed as an extension. TD3 takes into account the interaction between function approximation error in both policy and value updates and uses target policy smoothing and policy delays [3].

In this paper, our goal is to develop and utilize an implementation of TD3 that efficiently converges to an optimal steering policy that allows a mobile robot to follow the path in the center of a simulated car racing track environment. As our approach is based on the TD3 algorithm, our modifications are made to the architectures of the actor and critic networks, the design of the reward functions, and the specific methodology of calling the training function during the learning process. The policy training is done by keeping the speed of the car constant for every episode. As an extension, the race car is trained with differing constant speeds for each episode. Considering that a lot of research in RL has been done on policy control and path planning, our paper adds to these active topics of research. Our codebase can be found on GitHub by clicking on the following link: Path-Following-Task-For-Race-Car-Robot.

2 Related Works

PID and RL have been leveraged to tackle many tasks in mobile robotics, such as path-following, wall-following, and path planning. For example, a robust PID controller has been used for the path-tracking task of mobile robots where the proposed approach was to use a simple linearized model of the mobile robot consisting of an integrator and a delay [13]. Furthermore, over recent years, RL has become an increasingly popular method in solving path-planning and obstacle-avoidance tasks [7]. RL creates agents that adapt to their environment, thus making RL more desirable than other control models in situations where the specific string of actions required to achieve the best performance is unknown [8]. One of the biggest advantages of RL over PID is that agents can develop optimal control policies without the need of manual intervention for parameter tuning [9].

Past research has used RL to solve path-following tasks like ours. For instance, Wang et al. present a motion planning model that can navigate an environment safely without any prior knowledge [10]. They believe that existing work on the subject was unable to be generalized and had trouble dealing with complex scenarios. They were able to learn task by decomposing the arbitrarily complex navigation problem into sub-tasks. Huang et al. followed a similar method but utilized Q-learning instead [11]. Gao et al. used DRL to make a mobile robot plan its path in a known environment [12]. They combined the Probabilistic Roadmap path planning algorithm with Twin Delayed Deep Deterministic policy gradients to create a novel path planner in the DRL model [7].

In this paper, we employed a DRL model based on the TD3 algorithm and tuned the hyperparameters to better fit our specific task of maintaining the moving race car to follow a path in the middle (center) of the track. What sets our approach apart is that instead of focusing on the generalization of our task to different complex scenarios, we focused specifically on the optimization of the training algorithm strictly for the most basic path-following task: steering a robot with constant speed (our main task). In particular, our goal was to design our model and train it to get good steering policies that were converged to quickly. As our approach is based on the TD3 algorithm, our modifications are made to the architectures of the actor and critic networks, the design of the reward functions and the action space, and the specific methodology of calling the training function during the learning process. We present a reward function that addresses specifically the issue of oscillations in path following and provides great results.

3 Methodology

3.1 Presumptions and Generalisations

As we are concerned more with the analysis and utilization of TD3 for path-following, our research makes the following assumptions and generalizations:

- **Path planning is already implemented:** Our model does not generate a path; a predetermined path is given to the model from which it is required to determine appropriate control actions to follow that path. As path planning can be implemented using different strategies, such as Rapidly-exploring Random Trees (RRTS) or the A* algorithm, we isolated the model from this task.

  In our implementation, the predetermined path is simply the center of the track from which relative measurements are given to the model.

- **Reliable sensor data is given:** Our model is not responsible for taking and processing any sensor readings and expects reliable sensor data to be provided to it. As this can also be implemented with various other strategies such as filters including the Kalman filters and the Particle filters utilizing sensor-collected data, our model was isolated from this task as well so that it primarily focuses on control optimization.

  In our implementation, we use ground truth values for the car's position, direction, and velocity relative to the track defined extensively in the environment section.

- **Speed of the car is maintained to be constant:** Initially in our model, we trained the car at a single defined constant speed, for all episodes, to first tune our training algorithm for faster convergence. Then as an extension, the car was trained so that in each episode the car was set at random constant speeds within the range of values, which were normally distributed between min_speed and max_speed.

  In our environment, as the car can lose speed at instances such as after taking a turn, the speed is forced to be constant by setting the wheel rotational speed at every time step in the step function.

3.2 Environment

The first step in our research was to choose a simulation environment specific to our goal. We used OpenAI's Gymnasium as the simulator and made use of the car racing environment in the Box2D module. In our implementation, we make use of some of the functionality from the modified version of this environment from the University of Toronto's CSC2626 course's car racing environment as discussed in detail ahead (see "Acknowledgements" section in "Appendix" for repository link). The environment consists of a race track hosting a wide range of turning angles with grass tiles on either side of the track (Figure 1). The original environment was designed for an imitation learning task that aimed to train the race car to complete laps around the track. We modified the environment to match the specifications of our problem.

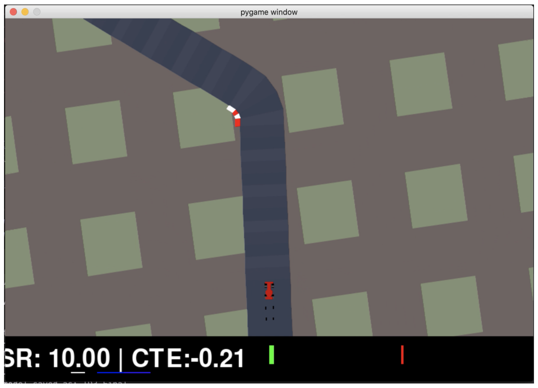


Figure 1: Environment (zoomed out to show turn). SR: Step Reward | CTE: Cross-Track-Error

We defined the **observation space** to be continuous consisting of the following:

- **The direction of the car relative to the direction of the track (Error Heading):**
  The direction of the car can be defined as the angle between the car frame and the world frame. Similarly, the direction of the track is the angle between the tangent to the center of the track and the world frame. Subtracting these two angles gave us the direction of the car relative to the direction of the track. The implementation for this task was taken from the University of Toronto's CSC2626 course's assignment 1. The function is defined in the $env.py$ file as $get\_cross\_track\_error()$.

- **The cross-track error (CTE):**
  The cross-track error in our problem is defined as the smallest distance between the car and the track center which is signed with respect to which side of the track the car is on relative to the car frame. It can be formalized as $c(t) = (x_t - x_d)$, where x_t is the current position of the car in the track frame and x_d is the nearest point on the center line as the origin of the track frame. In our implementation, it is calculated by first finding the closest point on the track from the car as a vector in the world frame along with the position of the car as a vector in the world frame. Then the resultant vector from the difference between these two vectors is transformed from the world frame to the track frame which gives us our cross-track error as the x-component of this transformed vector. For consistency in our definition in the rest of our implementation, we take the negative of the x-component as our CTE. This is also defined in the same function mentioned above: $get\_cross\_track\_error()$.

- **The speed of the car in the world frame:**
  This observation is added only in the case where the model is being trained for various constant speeds, or acceleration added to the action space, so that instead of just learning for a single constant speed, the model can eventually generate steering policies that can learn to relate the car's speed with the rate of steering control required.

The **action space** is also continuous and only consists of the angular rotation of the vehicle which was bound between values [-1, 1]. In this task, a value of -1 implied a full turn to the left and a value of 1 implied a full turn to the right. As an extension to our research, we plan on adding acceleration and brake to the action space as well, however, in this paper, all of the research is performed with only steering in the action space.

Normalization of Observation Space: As our model architecture defines our actor-critic networks as neural networks, the observation space was normalized to be capped between -1 and +1, the same as our action space, to ensure a consistent scale that is known to help improve the convergence and performance of neural networks.

The method for normalization was straightforward for the direction of the car and the cross-track error (CTE) where each of them was divided by their maximum values: π and w respectively, where w is the half-width of the road.

To normalize the speed, between min speed and max speed, to [-1, 1], it was first noted speed will always be positive. Hence, the range would be [0, 1]. However, an edge case here is when the speed is equal to min speed, its normalized form would be 0. In terms of the neural network multiplying this value within it somewhere to increase or decrease the rate of turn, this could be problematic at min speed which is not 0. Hence, the normalization range was considered to be [0.01, 1]. Conceptually, we ensure speed can never be zero.

This was achieved using the following formula:

$$X_{\text{normalized}} = \frac{X - X_{\text{min}}}{X_{\text{max}} - X_{\text{min}}} \times (\text{new\_max} - \text{new\_min}) + \text{new\_min}$$

3.3 Reward Function Design

The reward function was designed and optimized for quick convergence as defined below followed by explanations for the respective design choices. In reinforcement learning, it is generally recommended to have a dense reward policy, such that the agent receives some reward frequently, preferably at each time step. For our task, we define this reward function as a piece-wise function handling three different, but not mutually exclusive goals:

1) The car is maintained to be close to the path (the center of the track): The model is rewarded positively for keeping the car on the path with a fixed maximum reward that decreases exponentially as the square of the cross-track error is subtracted from this maximum. This design choice of shifting the negative squared CTE is defined in detail in the next tuning section. $[R_{shift} - c(t)^2]$

2) The car is prevented from forming oscillating patterns: The model is penalized for oscillations around the path that are captured using the variance over a fixed number of previous cross-track errors: N_{prev}. The variance is re-scaled by a factor of V_{scale} and subtracted from the reward as the penalty. $[V_{scale} * Var(c(t)_{N_{prev}})]$

3) The car is prevented from going too far from the path: In our implementation, the car is penalized a fixed large number, OFF_ROAD_PENALTY, and the episode is terminated when the car's center crosses the edge of the road.

Combining all three components, the complete reward function for a given action is given by the following:

$$R(a, s) = \begin{cases} [R_{shift} - s[1]^2] - [V_{scale} * Var(s[1]_{N_{prev}})] & s[1] \leq w \\ \text{OFF\_ROAD\_PENALTY} & s[1] > w \end{cases}$$

In the above expressions, s[1] corresponds to the CTE c(t) which is the second state in our implementation, w is half the width of the track. The rest are constants defined in our code as

$$R_{shift} \equiv \text{REWARD\_VSHIFT}$$
$$V_{scale} \equiv \text{VARIANCE\_RESCALE}$$
$$N_{prev} \equiv \text{NUM\_PREV\_ERRORS}$$

and OFF_ROAD_PENALTY as is.

Initially, our reward function was simply based on goal [1] where the reward was negative of the square of the CTE, $-c(t)^2$, and episode termination was either completion of a lap or reaching the allotted max time steps. The negative squared CTE was selected as the reward function as its gradient was a linear functions as shown in Figure 2. This gradient is important as it defines the reward signal the model receives when the car changes its position horizontally relative to the direction of the path. A linear signal is easily interpretable by the back-propagating neural networks in our TD3 actor-critic model architecture.

However, with this basic setup of the reward function and simulation termination conditions, the initial episodes always took the maximum number of time steps staying at a poor policy in the initial training iterations. Apart from the increased convergence time, when the model did converge to better policies, they always had a lot of oscillating patterns as the car always overshot the path.

Over time, the model did learn to get stable but it took longer to land on a decent non-oscillating policy. Hence, to cap the initial episode time where the car was not following the path and went off-road, the OFF_ROAD_PENALTY was introduced and episodes were terminated on going off the track. This made convergence to a decent policy to be done in minutes with a few minor adjustments described in the next section.

Next, our goal was to counter learned oscillations more explicitly; the maximization of the negative squared CTE also did this implicitly as the model would eventually learn that maintaining the car to be consistently at the center earned the highest rewards. However, one issue with the squared CTE was that if the model did oscillate with a high frequency and low amplitude close to the center, the reward remained high. To make our model grasp the concept that any sort of oscillations or erratic movements are bad policies more efficiently, the car was penalized for actions that resulted in these movements. The notion of oscillations and erratic changes was captured by the variance of the CTE.

The variance of the CTE was calculated over a window and subtracted from the reward. Using some manual trials, N_{prev} number of previous errors was determined as the size of that window. The variance also needed to be re-scaled as discussed in the next section.

3.4 Reward Function Tuning

For the previous design changes to be usable by the model to converge faster, the model had to associate the rewards and penalties with the actions efficiently; there needs to be a balance between the rewards and the penalties so that high rewards/penalties don't mask low rewards/penalties. For this reason, the constants defined above to re-scale the rewards and penalties were introduced.

Firstly, the OFF_ROAD_PENALTY was fixed as a high number: 100. As we want the car to change its policy quickly to avoid going off-road, this penalty was a big number so that the backpropagation gradients can be steep for the model to quickly update its weight and understand what is considered as a definite bad policy. As this was a fixed penalty, it was decided that the rest of the rewards should be relative to this.

With the reward function as just the negative squared CTE, one of the issues following a large off-road penalty was that the model tended to avoid an even larger penalty accumulating (which it gets if it was not exactly at the dead center of the road) by simply terminating the episode going off-road as soon as possible. To counter this, a vertical shift by +1 was included to push the negative squared CTE above the x-axis so that, instead of always penalizing for being away from the center, a positive reward was given for being close to the center. This way the model could associate actions that kept the car on the road as part of a good policy and going off-road as the problem. In our implementation, we use a normalized CTE that ranges between -1 and +1 instead of -w and +w, where w is the half-width of the road. Hence, shifting this negative squared CTE by 1 ensures its range is between 0 and 1, entirely positive. Once, where all the rewards being negative blurred the actions and their respective rewards together, now, this change creates a contrast that allows the model to have more clarity in associating rewards with actions.

As the variance of the CTE was a small number compared to the square of the CTE, for its tuning it was scaled by a factor of 20 to have comparable values to the positive reward. Hence, if there were too many oscillations, subtracting the variance of the CTE from the reward made it either close to 0 or negative.

Now, with these changes, the model converged to a very stable policy very quickly. To further exaggerate the convergence, it was predicted that overall high rewards may give steeper gradients that will initially train the random weights quickly to a decent policy. As this prediction reflected in our trials, all the reward constants were scaled by a factor of 10. Altogether, the following tuning gave the best results for our reward function:

$$\text{REWARD\_VSHIFT} = 10$$

$$\text{VARIANCE\_RESCALE} = 200$$

$$\text{OFF\_ROAD\_PENALTY} = 1000$$

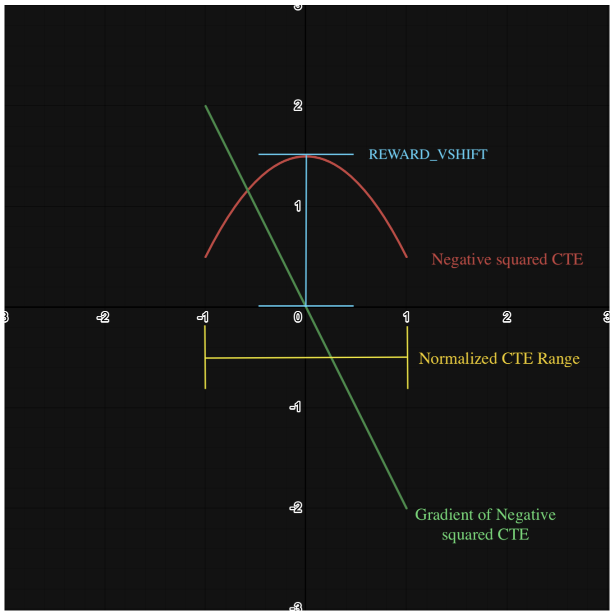


Figure 2: Negative squared CTE reward (without the overall factor of 10)

With these reward-shaping changes, we were able to achieve a dense reward policy with the reward at each time step precisely guiding our model on what to do and what not to do.

3.5 TD3 Algorithm and Training

The nature of the problem required continuous action and state spaces which poses a challenge for some traditional reinforcement learning methods. Hence, we took inspiration from TD3 algorithm, which is specifically designed for continuous action spaces [3]. The input dimensions for the actor-critic neural networks are made of the observation space dimensions and action space dimension, both of which are relatively very small numbers after our modification to the environment; in our implementation, we scaled the layer sizes down to (100, 100) for both the actor and critics networks from the original sizes of (400, 300) recommended in the paper [3]. We found that this reduced training time while keeping the performance the same. Most of our modifications revolved around implementing different training techniques while keeping the original TD3 algorithm mostly the same.

Vectorization of environments: For this task, we made use of a variety of different training techniques to speed up training and to achieve better results. In our problem, training the networks took significantly less time than collecting data from the episode simulations. As our bottleneck was episode simulation, it was realized that it is possible to use multiple CPU cores to run parallel environments. Although parallelization with multiple CPU threading may still slow down each vectorized instance of the simulation compared to a single simulation, the data collection trade-off was worth the extra time. If we ran three simulations in parallel, on our machines, we could collect three times more data with hardly a 10 percent increase in time.

The implementation for this vectorization was followed straight from the OpenAI Gymnasium's vector class. The vectorization ran 3 environments in parallel where each environment, initialized on different seeds, ran independent episodes using the current common policy. The difference in seeds allowed the environments to run on unique tracks from each other. As soon as an episode is terminated, the respective environment auto-resets to start a new episode. The condition to start training and end data collection in an iteration is that the initial three episodes in each environment are finished. This decision was taken so that if a car continued to do well on one of the environment's initial episodes due to a policy's strength, it was not stopped by the other environment's episodes terminating early due to that policy's weaknesses. All of the data regarding a policy should be collected. Hence, with 3 parallel environments, we would have a minimum of 3 episodes but not necessarily a maximum of 3.

By running agents in parallel environments, we were able to greatly reduce the time it took to collect data which, consequently, reduced the time to convergence as each training iteration had more data to learn from.

Step reduction in exploration noise and Tau(τ): We added random noise distributed $N(0, 0.01)$ to the actions at each time step to encourage exploration of different states. However, after the rewards reach a certain fixed threshold determined using manual trial observations, we reduced the exploration noise by 50% to encourage the model to exploit the current decent policy instead of exploring further. As our model has to learn to output a single action, it does not have to explore how different combinations of actions

Along similar lines, it was observed that our model tends to learn a good policy and then eventually diverges to bad policies which it is unable to recover from. Hence, it was also theorized that as our model is in the process of converging to a good policy, its target network updates overshoot the optimal policy. To stabilize these updates as our model approached closer to good policies, the target network update control hyper-parameter Tau τ was also lowered from 0.001 to 0.00075 after our model reached another predetermined reward threshold. This enables the model to take more stable and precise update steps for the actor and critic networks as it begins to converge closer to a good policy.

Standardized learning iterations: One challenge we encountered during training was that at times, the model would adopt a policy where the car would veer sharply right or left off-road right at the beginning of an episode, quickly terminating it within a few time steps. Since these time steps serve as training iterations for the TD3 train function, the model struggled to adjust its actor-network weights to adopt a different policy with low training iterations, leading to a loop of continuously veering off-track without converging.

To address this issue, we implemented a solution wherein, if the number of time steps fell below a certain threshold, a standard number of training iterations were passed to the train function. After experimentation, we found that setting this threshold at 500 effectively broke the loop and allowed the model to converge.

3.6 Extension: Training for Various Constant Speeds and Acceleration

Various constant speeds For clarity, various constant speeds in this context can be defined as setting differing constant speeds for each episode. The speed within each episode remains constant and is set at random constant speeds within the range of values, which are normally distributed between min_speed and max_speed in the variable speed settings dictionary defined as an environment attribute in our implementation.

Our expectation from training with varying constant speeds is that the model can eventually generate steering policies that learn to relate the car's speed with the rate of steering control required for various speeds. As learning steering acceleration and braking together takes longer to converge, this implementation also allows us to take a step towards curriculum learning where first we learn to steer at different speeds independently. This should allow the model to handle steering at any speed even if it changes within the episodes. Hence, one aspect of acceleration would be sorted out making the rest of the learning required a little easier. This is discussed further in the limitations section.

Acceleration For the acceleration, in addition to the steering, we added two more actions to the action space: brake(slow-down or deceleration), and gas (speed-up or acceleration). For both brake and gas respectively, the values for action are between 0 and 1.

We can define training the model as finding the optimal policy that allows optimal actions (values) for steer, gas, and brake for every time step for each episode. It is important to note that the goal is still to make sure that the race car follows the path on the track by remaining in the middle of it. Therefore, the appropriate steer, gas and brake actions that ensure that will be considered as the optimal ones.

During training, the real challenge when considering acceleration is not only to optimize for maximum reward for just staying in the center but also to account for other factors introduced by brake and gas. For example, if the car stays at the center with no speed it will maximize the reward, however, this trivial solution does not amount to anything. Instead, the car should use brakes only for making turns and should prioritize using gas for other straight regions on the track. As there are just a few regions corresponding to turns in the track, the car must prioritize using gas more than using brakes. Another important aspect that must be considered when updating the reward is the speed of the car itself. Considering that the speed of the car affects its maneuverability, the reward function should take speed into account for quicker convergence. Hence, the reward function of the car penalizes very high and very low speeds. These rewards/penalties add to the existing reward function made for steering.

$$R_{acceleration}(a,s) = R(a,s) + R_{addition}(a,s)$$

$$R_{addition}(a,s) = \begin{cases} [a[1] - a[2] * B_{scl}] + [(1 - |s[2] - S_{opt}|) * E_{scl}] & s[2] \geq 0.1 \\ [a[1] - a[2] * B_{scl}] + [(1 - |s[2] - S_{opt}|) * E_{scl}] - P_{spd1} & s[2] < 0.1 \\ [a[1] - a[2] * B_{scl}] + [(1 - |s[2] - S_{opt}|) * E_{scl}] - P_{spd2} & s[2] < 0.05 \end{cases} \quad 341$$

where a[1] is the gas action, a[2] is the brake action and s[2] is speed state. The rest are scaling constants that were tuned in a similar way to the original reward function (our main task). Here are the tunings:

$$Braking\ action\ rescale: \quad B_{scl} = 20$$

$$Optimal\ speed: \quad S_{opt} = 0.5$$

$$Speed\ error\ rescale: \quad E_{scl} = 10$$

$$Low\ speed\ penalties: \quad P_{spd1} = 25, \quad P_{spd2} = 1000$$

So, we proportionally reward for the gas action between 0 and 1, proportionally penalize for the brake action between 0 and B_{scl}, and reward for speed being close to the speed S_{opt} considering speed is normalized. We also penalize in steps for low speeds with penalties P_{spd1} and P_{spd2}.

4 Evaluation

The performance evaluation of the robot was conducted using a range of metrics, both quantitative and qualitative. While qualitative performance was evaluated by visually confirming the stability and distance of the evaluated episode, the quantitative metrics provided a deeper insight and allowed for a greater level of precision in policy evaluation. The quantitative metrics used are summarized below.

4.1 Quantitative metrics

- **Average Cross-Track Error per Time step:**

  The primary quantitative metric for evaluation is the average cross-track error, which measures the deviation of the vehicle's path from the center of the path. This metric provides us with crucial information of the average distance that the vehicle was from the desired path. To ensure a comprehensive assessment, the average cross-track error was calculated over 10 episodes each with distinct tracks.

- **Trajectory Smoothness and Vehicle Stability:**

  The smoothness of the vehicle's trajectory is crucial to identifying any jerky, erratic, or oscillating movements. This metric is determined by analyzing the variance of the cross-track error over 10 episodes. The variance provides a metric for the stability of the vehicle as it attempts to follow the path. As with the average cross-track error, this analysis will be performed over multiple intervals to provide a comprehensive understanding of trajectory smoothness.

- **Average Reward:**

  For this metric, we considered both the average episode reward and the average reward per action over all 10 simulated episodes. The average episode reward provided us with a good idea of the quality of the policy and whether the vehicle traversed the entire track.

  Although this metric could differentiate between 'good' and 'bad' policies, it could not gauge how 'good' a 'good' policy was. For instance, policies that had high-frequency low amplitude oscillations accumulated greater rewards than stable policies that were at a slight offset from the center due to them taking more actions to complete the track.

  This drawback is accounted for by the reward-per-action metric, which provides information on how good each action is on average. As per the design of our reward function, the maximum reward per action is 10, hence values closer to 10 indicate that the policy is able to stay close to the path while maintaining stability.

- **Algorithm Convergence Rate:**

  Evaluation of the learning algorithm's performance will involve assessing its convergence rate towards satisfactory policies. Faster convergence indicates more efficient learning, which is crucial for real-world deployment. The algorithm's convergence rate will be compared against baseline models such as TD3, DDPG, and PPO to gauge its effectiveness and efficiency. This information will be presented in a number of graphs below.

The learned policy was able to achieve great stability while following the required path. These results are reflected by the above metrics, which are displayed in Table 1. The metrics provided are from the tenth training iteration (Policy_10_actor.pth and Policy_10_critic.pth on Github), which yielded the best performance both quantitatively, and qualitatively.

Variance of CTE	0.010
Average reward	18698.445
Average reward per timestep	9.917
Average tile reward:	63.396
Average CTE	0.179

Table 1: Quantitative Results of Learned Policy for the Main Task

4.2 Comparisons to DDPG, PPO, SAC

Given that we were able to achieve great results on our task, we decided to evaluate our policy against other state-of-the-art Reinforcement Learning methods. The comparison was designed to reveal the efficiency with which each method trained and the performance they were able to achieve on the given task. Each method was trained to learn the same task using the same environment and reward function. Each algorithm was trained for approximately 50000 time steps, with some executing more episodes than others during that time frame due to differences in episode lengths while training. Keeping these variables the same for all algorithms, we chose a set of comparisons for evaluation. The comparison metrics were the following:

- **Reward vs Timesteps:** The rewards during training were recorded and plotted against the number of timesteps for each of the algorithms.

- **Reward Training curve:** The episode rewards during training were recorded and plotted against the number of episodes for each algorithm.

- **Episode Length Training curve:** The episode lengths during training were recorded and plotted against the number of episodes for each algorithm. This graph shows how much of the track the vehicle was able to traverse before going off-track for each episode during training.

Our implementation of the TD3 algorithm was able to outperform the other algorithms in each of the training metrics. This was expected as our algorithm was precisely fine tuned to meet the specifications of the given task, where as all of DDPG, PPO, and SAC were trained using the default parameters provided in the Stable-Baselines3 implementation. Figure 3 displays episode rewards plotted against the number of time steps. Figures 4 and 5 show the episode reward curve and episode length curve plotted against the number of episodes, respectively, for the full 50000 time steps. Note that some models achieved longer episodes earlier in the training, hence they reached 50000 timesteps in a lower amount of episodes. Figures 6 and 7 show the episode reward curve and episode length curve, respectively, for the first 48 episodes, which is when our TD3 algorithm reached 50000 timesteps. Figures 6 and 7 show the training progress of our implementation of TD3 as compared to DDPG, PPO, and SAC at the same number of episodes.

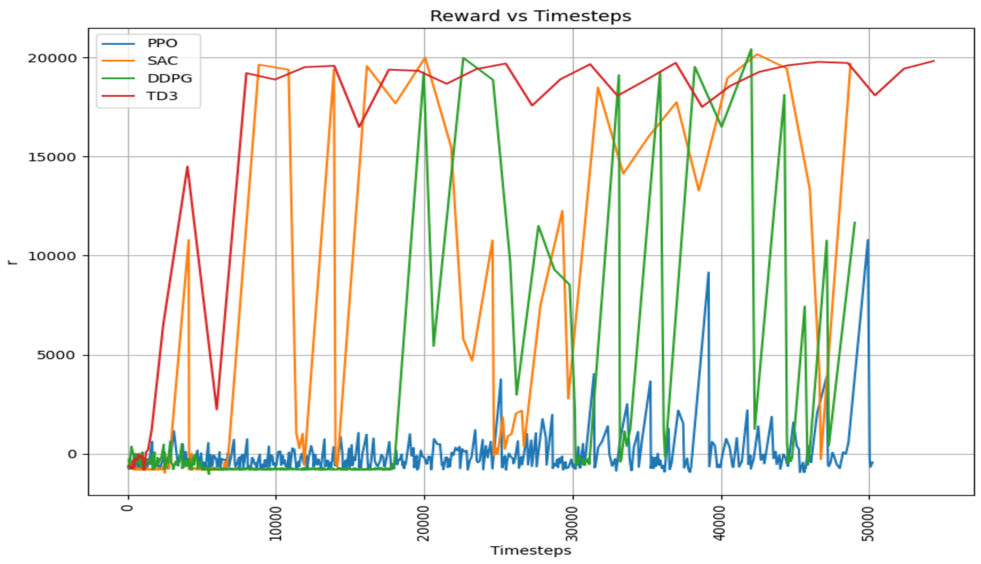


Figure 3: Reward Training Curve (Against Timesteps)

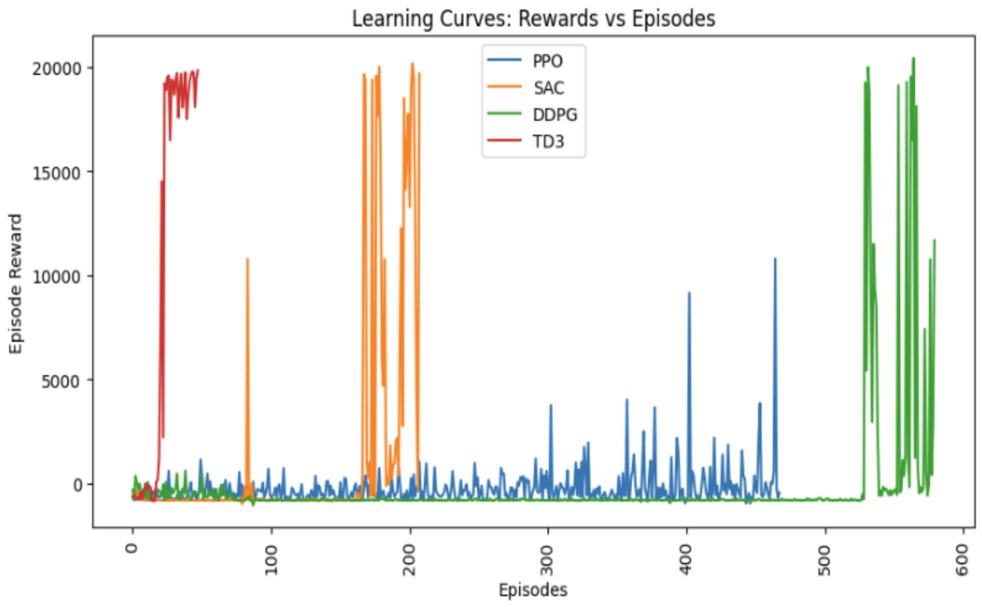


Figure 4: Reward Training Curve (Against Episodes)

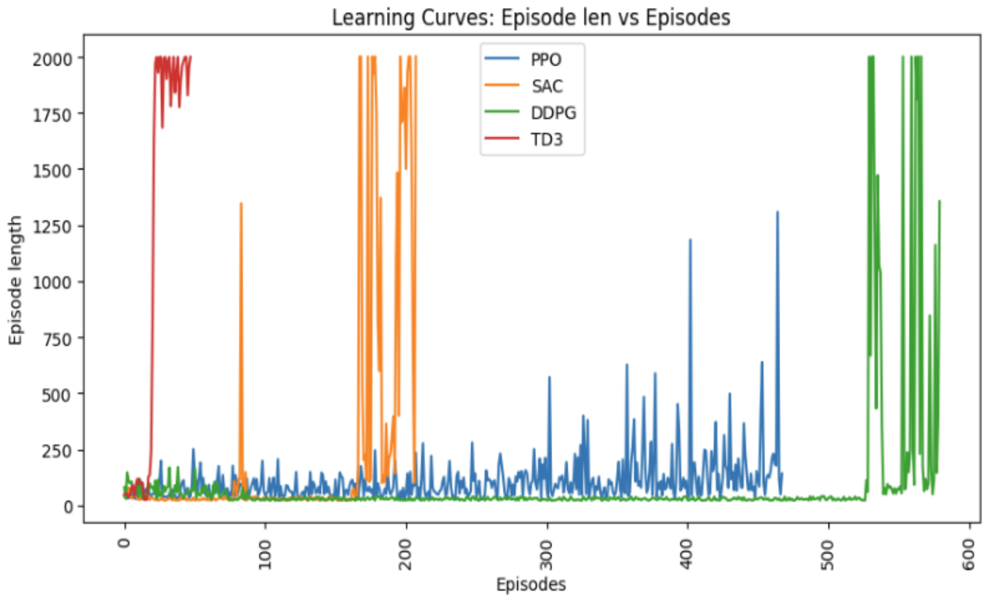


Figure 5: Episode Length Training Curve

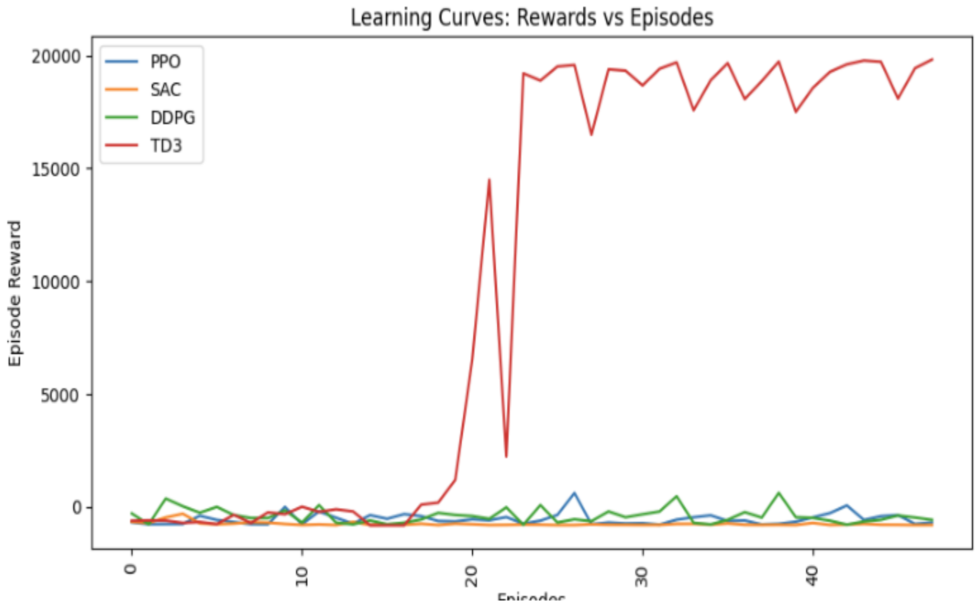


Figure 6: Reward Training Curve (First 48 Episodes)

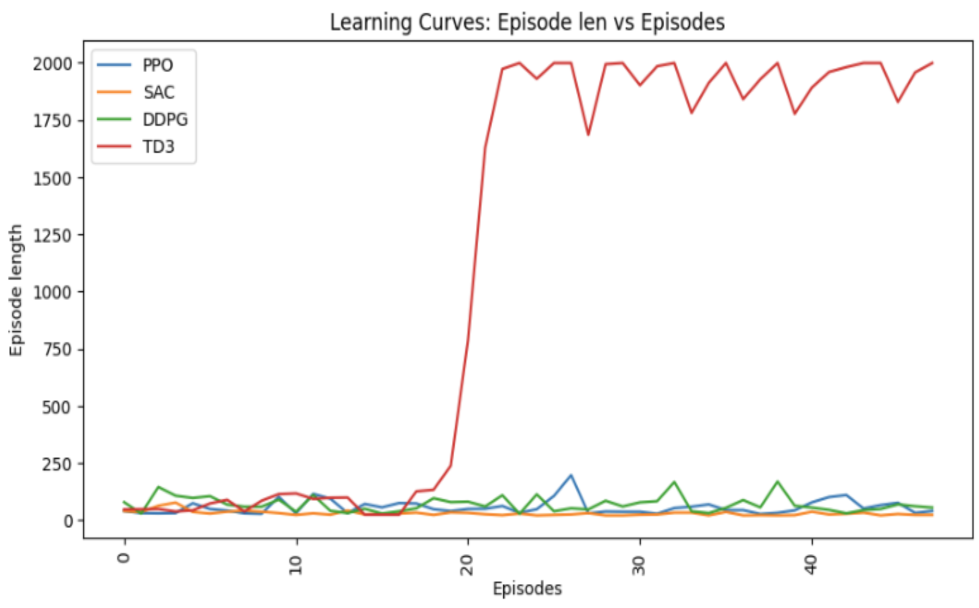


Figure 7: Episode Length Training Curve (First 48 Episodes)

The steps to reproduce all results, and to run the training code will be given in the Github README.md file.

4.3 Results Regarding the Extensions to Our Main Task

As mentioned before, for our main task, we only consider the constant speed of the car. However, for our extensions, we consider the various constant speeds as well as the acceleration (gas and brake) actions for the race car as it follows the path on the track by staying in the middle of it. Therefore, here are our results for these extensions.

4.3.1 Various Constant Speeds

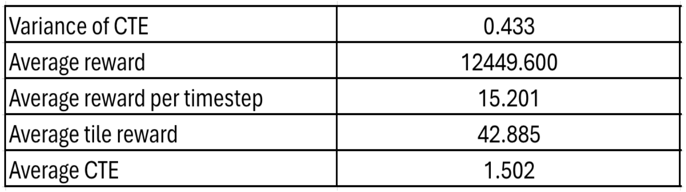
After testing, we were able to get good results for this task as well. The model did not oscillate and performed well by generalizing for different speeds. For qualitative analysis, These can be visualized by running the test file and using the best policy found for this case as described in the README.md file. The quantitative results for this task are as follows:

Variance of CTE	0.084
Average reward	16810.106
Average reward per timestep	9.784
Average tile reward:	58.066
Average CTE	0.585

Table 2: Quantitative Results of Learned Policy for Variable Speed

4.3.2 Acceleration

After testing, we were not able to get really good results for this task. The car did oscillate and go off track at times. These can be visualized by running the test file and using the best policy found for this case as described in the README.md file which can give a better qualitative understanding. The quantitative results for this task are as follows:



Variance of CTE	0.433
Average reward	12449.600
Average reward per timestep	15.201
Average tile reward	42.885
Average CTE	1.502

Table 3: Quantitative Results of Learned Policy for Acceleration

5 Limitations

By analyzing and examining our approach, we notice that we could have achieved better performance and efficiency specifically during the model's training when considering the acceleration (gas and brake) of the race car as the two additional introduced actions. However, in both constant and varying constant speed, we have reached a plateau where not many improvements are left to be made.

5.1 Acceleration

Considering the added complexity associated with this task, our approach does not update the reward in a way that allows training to be smooth and efficient and to converge relatively fast.

In the future, we can improve upon these limitations and challenges when considering the acceleration of the race car by implementing an approach known as curriculum learning. We can break the task into segments like steering and acceleration and then train the model in steps so that the complexity of our task gradually increases.

We have successfully implemented training for differing constant speeds. Our model should have learned a steering policy that can relate the car's speed with the rate of steering control required. Hence, essentially steering for any speed within the range of speeds trained is possible even if the speed changes during an episode. During this phase, the model should mask the acceleration actions(gas and brake) and introduce them later on.

Then, after the steering skill is acquired, the model's focus should shift to finding the optimal speed at different sections of the track with gas and brake unmasked. Although the model would still need to understand the relation between the new actions and steering, this should theoretically reduce the initial overhead where the model is unable to relate the effects of the three actions themselves to the rewards acquired.

6 Conclusion

In conclusion, our research presents an approach to solving a path-following task tailoring the utilization of the TD3 algorithm for a faster convergence specific to our task. By making modifications to the usage of this algorithm and determining a reward function, we were able to achieve an optimized algorithm that trains a model with quick stable convergence to follow a given path with great stability and efficiency.

Through our methodology, we described the importance of reward function development, environment setup, and training techniques to optimize the learning process. In our implementation, our reward function provides dense signals to the vehicle to remain close to the middle of the track and to not oscillate outside of it.

There are still certain limitations that need addressing to improve our implementation. The biggest limitation is how often the policy would get stuck while training specifically for acceleration. Although we managed to improve this by adjusting some hyper-parameters, stagnation still occurred. More research is required on utilizing the curriculum learning approach to assess its effectiveness.

Overall, the main goal of this research was successfully achieved. Our research adds to the vast knowledge of RL for robotics. By addressing the key challenges, we hope to help others who want to develop similar algorithms so that they can take our work a step ahead.

7 Appendix

7.1 Acknowledgements

University of Toronto's CSC2626 Assignment 1 repository:
https://github.com/florianshkurti/csc2626w22/tree/master/assignments/A1

7.2 Contributions

Note: due to a problem with the difference in commit emails, the contributions section on GitHub does not show everyone's work properly. Please refer to the commits in the main branch directly if required.

Hadi Hafeez

Code:

- vectorizedMain.py
- env.py (Modifying for our env, designing and tuning the reward functions)
- TD3.py (vectorization of select action)
- benchmarks.py
- test.py
- README.py

Report:

- Methodology
- Evaluation
- Limitations
- Related works
- Introduction

Hamza Faisal

Code:

- main training loop (non-vectorized version, base code)
- Actor.py
- Critic.py
- utils.py (Replay Buffer)
- Hyperparameter tuning

Report:

- Methodology
- Evaluations
- Related Works
- Limitations

Mahan Mohammadi

Code:

- TD3.py(implemented the TD3 algorithm and the methods for it)
- Critic.py (implemented my own version that got then corrected)
- utils.py (made some corrections and cleanings to the file)
- env.py (made modifications for the step function, action space and acceleration (gas and brake) actions for the race car)
- vectorizedMain.py (made some polishing, some additions to the argument parser, hyperparameters tuning and added some conditionals regarding the acceleration of the race car)
- README.md (made improvements and useful modifications)
- requirements.txt
- .gitignore

Report:

- Abstract
- Introduction
- Related Works (proof-reading and added content regarding some other similar research papers related to our topic)
- Methodology (Proof-reading)
- Evaluations (Added the plots, contributed to the description of plots and tables, contributed to proof-reading and wrote the section **Results Regarding the Extensions to Our Main Task**)
- Limitations (Contributed to it with Hadi equally as we were working on it together)
- Conclusion (Proof-reading)

Santiago Garrido

Code:

- Added README.md
- Tuning Hyperparameters for our approach
- Researched related code bases to create a background for our approach
- Clarified sections of the codebase

Report:

- Related Works
- Conclusion
- Proof-read the report
- Corrected parts of Introduction
- Reference

To conclude, for this project, everyone contributed equally, and we worked as a unit to do all the tasks for this project.

8 References

[1] Davide Di Febbo, Emilia Ambrosini, Matteo Pirotta, Eric Rojas, Marcello Restelli, Alessandra L. G. Pedrocchi, and Simona Ferrante. Does Reinforcement Learning outperform PID in the control of FES-induced elbow flex-extension?.*Axelgaard Manufacturing*. https://www.axelgaard.com/Docs/MeMeA2018_diFebbo.pdf

[2] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Hess, T. Erez, Y. Tassa, D. Silver, and Daan Wierstra, Continuous control with deep reinforcement learning, *International Conference on Learning Representations (ICLR)*, 2015, 1509.02971, arXiv, cs.LG. https://doi.org/10.48550/arXiv.1509.02971

[3] S. Fujimoto and H. van Hoof and D. Meger, Addressing Function Approximation Error in Actor-Critic Methods, 2018, 1802.09477, arXiv, cs.AI. https://doi.org/10.48550/arXiv.1802.09477

[4] Lakhani AI, Chowdhury MA, Lu Q. Stability-preserving automatic tuning of PID control with reinforcement learning. Complex Engineering Systems. 2022; 2(1): 3. http://dx.doi.org/10.20517/ces.2021.15

[5] V. R. Konda, J. N. Tsitsiklis, Actor-Critic Algorithms, *NeurIPS*, https://proceedings.neurips.cc/paper_files/paper/1999/file/6449f44a102fde848669bdd9eb6b76fa-Paper.pdf

[6] Behnaz Hadi, Alireza Khosravi, Pouria Sarhadi. Deep reinforcement learning for adaptive path planning and control of an autonomous underwater vehicle. *Applied Ocean Research* 2022, 129, 103326. https://doi.org/10.1016/j.apor.2022.103326

[7] K. Almazrouei, I. Kamel and T. Rabie. Dynamic Obstacle Avoidance and Path Planning through Reinforcement Learning. Applied Sciences, 13(14):8174, 2023. https://doi.org/10.3390/app13148174

[8] T. Johannink, S. Bahl, A. Nair, J. Luo, A. Kumar, M. Loskyll, J. Ojea, and E. Solowjow and S. Levine. Residual Reinforcement Learning for Robot Control. 2019 International Conference on Robotics and Automation (ICRA), 2019. doi:10.1109/icra.2019.8794127

[9] C. Chen, S. Jeng and C. Lin. Mobile Robot Wall-Following Control Using Fuzzy Logic Controller with Improved Differential Search and Reinforcement Learning. Mathematics, 8(8):1254, 2020. doi:10.3390/math8081254

[10] N. Wang, D. Zhang and Y. Wang. Learning to Navigate for Mobile Robot with Continual Reinforcement Learning. 2020 39th Chinese Control Conference (CCC), 3701-3706, 2020. 10.23919/CCC50068.2020.9188558

[11] L. Huang, H. Qu, M. Fu and W. Deng. Reinforcement Learning for Mobile Robot Obstacle Avoidance Under Dynamic Environments. PRICAI 2018: Trends in Artificial Intelligence, 441-453, 2018. doi:10.1007/978-3-319-97304-3_34

[12] R. Gao, X. Gao, P. Liang, F. Han, B. Lan, J. Li, J. Li, and S. Li. Motion Control of Non-Holonomic Constrained Mobile Robot Using Deep Reinforcement Learning. 2019 IEEE 4th International Conference on Advanced Robotics and Mechatronics (ICARM), 348-353, 2019. doi=10.1109/ICARM.2019.8834284

[13] Julio E. Normey-Rico, Ismael Alcalá, Juan Gómez-Ortega, Eduardo F. Camacho. Mobile robot path tracking using a robust PID controller. *Control Engineering Practice* 2001, 9, 11. https://doi.org/10.1016/S0967-0661(01)00066-1