Autonomous Control of an Ackermann Car Using Reinforcement Learning
ASEN 5264: Decision Making Under Uncertainty

Adam Buencamino
University of Colorado Boulder
SID: 109312450

Christopher Krebs
University of Colorado Boulder
SID: 108981667

Jake Miles-Colthup
University of Colorado Boulder
SID: 108951691

***Abstract*—This study explores the use of reinforcement learning (RL) algorithms to train an Ackermann car in a simulated track environment with obstacles. Three different levels of access to sensor data were tested, ranging from full supervisor access to onboard sensor data only. The RL algorithms used were Proximal Policy Optimization (PPO) and Deep Deterministic Policy Gradient (DDPG). The first case used a discrete action space, full supervisor knowledge, and PPO, the second case used a continuous action space, full supervisor knowledge, and DDPG, and the third case used a discrete action space, only onboard sensor access, and PPO. The results show that all three levels of access achieved success, with the PPO algorithm achieving optimal policy convergence after 19,500 episodes for the first case, 3,700 episodes for the third case, and the DDPG algorithm after 1,200 episodes for the second case. Interestingly, we observed that the continuous action space in the second case allowed for smoother and more efficient navigation compared to the discrete action spaces used in the first and third cases. The findings of this study suggest that RL algorithms can effectively train an Ackermann car in a simulated environment with obstacles, and that the choice of action space can have a significant impact on the car's performance.**

1. Introduction

TABLE 1. LEVELS OF SUCCESS

Level Of Success	Description
1	Use a RL algorithm to solve a continuous state space, discrete action space Ackermann car representation through a specified simple race course. The algorithm shall have supervisor access to the car's position, velocity, and orientation as well as onboard sensor data.
2	Use a RL algorithm to solve a continuous state space, continuous action space Ackermann car representation through a specified simple race course. The algorithm shall have supervisor access to the car's position, velocity, and orientation as well as onboard sensor data.
3	Use a RL algorithm to solve a continuous state space, discrete action space Ackermann car representation through a specified simple race course. The algorithm shall only have access to onboard sensor data.

The purpose of this project is to implement multiple reinforcement learning algorithms to solve a simplified race course for an Ackermann car representation.[1] In addition, the team has compared the results of two different algorithms to accomplish the first and second levels of success. As seen in **Fig.1**, the car starts in the top center of the track and must travel clockwise, on-path for a single lap while avoiding obstacles. The simulation runs in a Webots environment and uses the deepbots framework to step the environment state. The robot is controlled with external control scripts written in the Python language with access to the Webots Supervisor node which gives exact world knowledge that may not be available to a real robot.

In order to accomplish this, the state space has been defined as continuous. For the first and second levels of success, the state space includes onboard LiDAR sensors as well as true knowledge of the position and velocity from the supervisor. In addition, sequential waypoints along the track are defined, which are used in the reward functions as defined in §3.2.3 and §3.3.3. The first level of success specifies the car's completion of the race course using a reinforcement learning algorithm for a discrete action space, whereas the second level of success must solve over a continuous action space. The discrete action space is further defined in §3.2.2, with inputs controlling the speed and steering angle of the vehicle to specific values. The continuous action space is also within the domain of vehicle speed and steering angle, but can be set to any value within the vehicle's maximum constraints §3.3.2. The third level of success, similar to level one, also uses a discrete action space. However, the state-observation space does not take advantage of the supervisor, but rather only includes onboard camera image data and LiDAR sensor measurements.

To meet the first and third levels of success, the team implemented a Proximal Policy Optimization RL algorithm to solve the course over a discrete action space. PPO uses a clamped surrogate objective with a natural policy gradient. This algorithm is described in more detail in §4.1. To meet the second level of success, the team implemented a Deep Deterministic Policy Gradient RL algorithm to solve the course of a continuous action space. This algorithm concurrently learns a value function and policy using a neural network to approximate the optimal value and policy. [2] This works efficiently with gradient-based learning for the

policy. This algorithm is described in more detail in §4.2.

2. Background and Related Work

Reinforcement learning is the basis for the algorithms used to solve the simplified race course. RL works by training an agent by interacting with an environment through actions with no knowledge of transition probabilities to additional states, or knowledge of rewards. The agent is able to learn its state through observing the environment and obtains a reward based on its current state and action. Methods within RL are classified as model-based or model-free and are further classified as on-policy or off-policy. On-policy algorithms only learn from data obtained from the current policy whereas off-policy learns from the current and all previous policies' results. In addition, neural networks (NNs) can be used with reinforcement learning in what is classified as "deep RL." Model-based RL algorithms attempt to learn transition probabilities T and rewards R, then solve the learned Markov Decision Process (MDP) to obtain the optimal policy.

Model-free RL attempts to find the optimal value function $Q^*(s,a)$ or optimal policy π^* without estimating T or R. The optimal value function is given by Bellman's Optimality Equation:

$$U^*(s) = max_a(R(s,a) + \gamma\, \mathbb{E}_{s' \sim T(s,a)}[U^*(s')] \quad (1)$$

Where U^* is the optimal value that accounts for the immediate rewards R, as well as the expectation of future rewards.

Neural networks, used for deep RL, are differentiable, parameterized functions to map inputs to outputs. These networks are trained to minimize a loss function l_θ which gives the difference between the current and desired outputs. In creating a NN, multiple layers may be used. Each layer is given by 2:

$$x' = \phi(\vec{W}\vec{x} + \vec{b}) \quad (2)$$

Where ϕ is a nonlinear activation function, which is necessary to model complex behavior. Without a nonlinear activation function in each layer, outputs would be limited to only linear combinations of input data. In each layer, $\vec{W}$ and $\vec{b}$ represent weights and biases that are produced respectively. When creating a multi-layered neural network, all layers are interconnected to produce a single output from an input. [3]

Within RL, one of the main challenges is exploration versus exploitation. Agents that explore more will obtain a more comprehensive knowledge of its environment but will sacrifice the accumulation of reward. On the other side, agents that prioritize exploitation will act toward what it thinks will obtain the highest reward, while ignoring unexplored actions that may produce better results. Thus, in the implementation of an RL algorithm, one must take into consideration the various hyperparameters used.

Over a discrete action space, Proximal Policy Optimization has become a very popular algorithm because of its success and ease of implementation. PPO operates very similarly to the Trust Region Policy Optimization (TRPO). TRPO seeks to train a policy constrained by maximizing the objective function, subject to KL divergence between a new and old policy known as the "trust region." [4] PPO improves off of TRPO by modifying the objective to penalize larger changes in policy called the "clipped surrogate objective" to balance exploration and exploitation in the environment, avoiding issues caused by overly optimistic objective estimates. [3] Both TRPO and PPO use a NN in training an agent's policy toward maximizing their respective objective functions. The clipped surrogate objective function used by PPO has been shown to achieve greater average rewards than objective functions without clipping or penalty, as well as the fixed KL divergence used by TRPO. More detail and results about PPO can be found in §4.1 and from Schulman et al. [5]

Another key challenge in RL is navigating within continuous action spaces. There are several algorithms that function well within discrete action spaces, such as Deep Q Networks (DQN), that quickly become computationally expensive with higher-dimension action spaces. DQN is a popular deep RL algorithm that has been implemented to achieve at or above human-level performance in several video games. In particular, DQN has been able to outperform several other RL algorithms on 49 Atari games and even performed at a human level in 43 games [6]. Similar to the Q-learning Rl algorithm, DQN trains an agent's policy to achieve an optimal value $Q^*(s,a)$, which is a function of a state-action pair given by Bellman's Equation 1. This works in an iterative manner where the agent selects a policy that maps states to actions and then updates the action-value function $Q(s,a)$, and the loss function with respect to an approximate target value of $Q(s,a)$. From there, a stochastic gradient descent can be performed to update the weights within the NN to train the agent's policy.

While very successful in the discrete action space of several Atari games, DQN becomes less viable with a larger, continuous action space. This is because it must evaluate the value function for each *discrete* action, which leads to dimensionality concerns, as the number of iterations increases exponentially based on the degrees of freedom.[2] Thus in order for DQN to be applied to a continuous action space, the actions must be discretized. This generates several concerns related to the level of granularity of discretization. A lower number of discretized actions may be less computationally expensive for an algorithm such as DQN, but this may hide essential information about the action domain.

Based on the lessons learned from DQN, an algorithm named Deep Deterministic Policy Gradient has been developed to learn policies in continuous action spaces. Similarly to DQN, DDPG trains a network off-policy using a replay buffer to avoid over-correlation between samples. In addition, both DQN and DDPG train a network to a target Q network. To avoid the issues of dimensionality and discretization of a continuous action space, DDPG employs an

actor-critic method based on a deterministic policy gradient. This algorithm bases actions on a deterministic policy for low-dimensional observations of the state. This algorithm has performed very well on multiple problems with continuous action spaces such as Cheetah, Cartpole, and the Torcs driving simulator. More details on this algorithm and results can be found in §4.2 and from Lillicrap et al. [2]

Related to this project of solving the race course, deep reinforcement learning methods have become very popular in improving the capability of autonomous driving. In order to implement a functioning RL algorithm for autonomous driving applications, several considerations must be made. These include what sensors should be used for the perception of the vehicle's surroundings, state space representation, decision-making algorithm implementation, and control of the vehicle. The formulation of the state and actions (discrete or continuous) helps guide the selection of an algorithm to implement for different autonomous driving functions and situations. Notably, PPO, TRPO, and DDPG have been popular algorithms implemented in simulations to serve as performance benchmarks. [7]

3. Problem Formulation

This section will detail the way that the team formulated the problem and will cover aspects that apply to all three level of success subproblems outlined in **Table. 1** as well as detail individual nuances in the following subsections.

3.1. Learning Environment

The different approaches discussed in this paper were all implemented to control the same Ackermann car vehicle. The important aspects of the Ackermann car to note are the control inputs. The Ackermann car is rear wheel drive in that a certain angular speed can be commanded to the rear wheels simultaneously. The car's direction is controlled by a steering linkage on the front two wheels where a steering angle is defined that turns the car in a circle about a radius r. These two inputs give the size of the action space for the RL algorithms. For the purposes of this paper no further knowledge about the Ackermann car is necessary however more information can be found in [1].

Success of the RL algorithm was determined based off of both successful completion of a simple race course as well as lap time on the race course. The race course consisted of a walled rectangular annulus with various obstacles as shown in **Fig. 1**

3.2. Discrete Action Space Formulation

For the first level of success problem, the team considered that the Ackermann car had state and action spaces as defined below.

3.2.1. State Space. The state space for this problem level encompasses both on-board sensors and supervisor knowledge of the environment. It consists of LiDAR beam measurements distributed evenly from 90 degrees to the left of the car clockwise around to 90 degrees to the right of the car, the car's position on the x axis, the car's position on the y axis, the speed of the car, and the angle of the velocity vector of the car with respect to the positive x axis of the world frame. This combination generates the continuous state space represented in **(3)**.

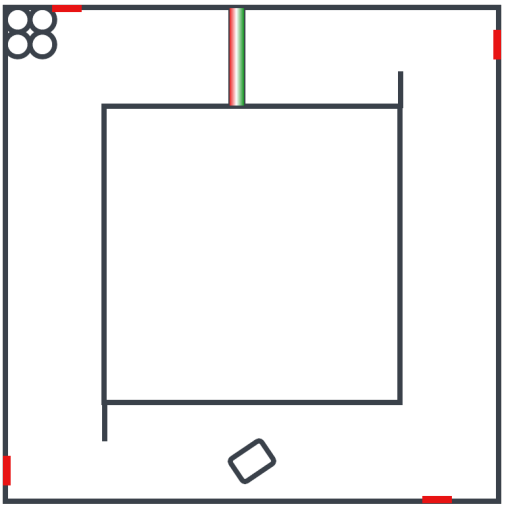

Figure 1. Race Course for Learning Problem

$$S = [d_{min}, d_{max}]^N \times [0, s_{max}] \times [-\pi, \pi] \tag{3}$$

Where d_{min} and d_{max} are the LiDAR sensor's minimum and maximum detection range, s_{max} is the maximum speed the vehicle is capable of, and N is the number of LiDAR beam measurements used, for the case of this problem, the team used $N = 60$

3.2.2. Action Space. For the second level of success problem, the team considered that the Ackermann car had a discrete action space, that is, only specific values could be applied to the car from the speed and steering angle domains. This action space is captured below in **(4)**.

$$A = \left[\theta \in \left\{i\frac{\theta_{max}}{10}\right\}\right] \times \left[V \in \left\{\frac{V_{max}}{2}, V_{max}\right\}\right] \tag{4}$$

For all i in [1, 10]

3.2.3. Reward. The reward function for this formulation was designed to incentivize the car to fully complete the track as well as to complete it as fast as possible. The reward function uses position-based waypoints to establish a reward for progress along the track, the track angle to ensure it is traveling in the right direction, and a series of checkpoints to help verify that the car is completing the track in the proper direction. The reward is terminated once the vehicle has collided with a wall (LiDAR rays read a small value). The reward function is described in **(5)**

$$R = R_{prog}k_h + R_{goal} \tag{5}$$

With components defined as

- R_{prog} = difference between closest waypoint index and current waypoint index
- k_h = normalized difference between track angle and car angle
- R_g = Reward equal to the number of discretized waypoints if the car reaches the end of track (completed all checkpoints) else 0.0

3.3. Continuous Action Space Formulation

For this level of complexity in the problem, the team considered that the Ackermann car had a state and action spaces as defined below.

3.3.1. State Space. For this problem, the state space was the same continuous state space used in the first level problem as defined by **(3)**.

3.3.2. Action Space. For this problem, the team considered that the Ackermann car had a continuous action space where a speed and steering angle could be applied at any value between the vehicle minimum and maximum constraints. This action space is represented below in **(6)**

$$A = [\theta \in [\theta_{min}, \theta_{max}]] \times [V \in [0, V_{max}]] \tag{6}$$

3.3.3. Reward. The reward function for this formulation was designed in a similar manner to that of the discrete action space case with the addition of a constant small negative reward to encourage speed as well as collision penalties. The reward is formulated as in **(7)**.

$$R = R_{prog} k_h + R_g + P_t + P_w \tag{7}$$

With components defined as

- R_{prog} = difference between closest waypoint index and current waypoint index
- k_h = normalized difference between track angle and car angle
- R_g = 100.0 if car reaches end of track else 0.0
- P_t = -0.04
- P_w = -2.0 if car collides with wall or obstacle else 0.0

3.4. Onboard Sensors Formulation

For the third level of success problem, the team desired to make the problem as representative of a real situation as possible. This meant that supervisor privileges were revoked and the state and action spaces were defined as follows.

3.4.1. State Space. The state space for this problem level encompasses only on-board sensors consisting of an RGB camera and a LiDAR sensor. For computational efficiency, the RGB camera state space contributions were greatly reduced. The RGB data was masked where each pixel represented binary information on whether or not the target object existed. This pixel data was then smoothed with a kernel in order to reduce the image size. This combination generates the state space represented in **(8)**.

$$S = [[0, 1]^{px \times py} \times [d_{min}, d_{max}]^{60}] \tag{8}$$

Where $px = 36$ and $py = 27$ represent the number of pixels in the x and y direction of the reduced camera frame and d_{min} and d_{max} again represent the minimum and maximum ranges of the LiDAR sensor.

3.4.2. Action Space. Due to the additional complexities of the state space, for this problem level, the team turned back to the discrete action space represented in **(4)**

3.4.3. Reward. The reward function for this formulation was designed to follow visual cues mounted on the walls of the rectangular course to reward both progression and course direction as shown in **Fig. 1**. The visual cues were represented by dark red rectangles located in each corner of the course. Once the vehicle has collided with a wall (LiDAR rays read a small value), rewards are terminated.

$$R = R_{\text{detect}} \tag{9}$$

- R_{detect} = If the RGB camera detects the visual cue and the front-facing lidar rays do not read a value (to encourage moving on to the next visual cue), the reward is 1. Else, the reward is zero.

4. Solution Approach

This section details the various learning algorithms that were implemented and applied to each level of success problem.

4.1. Discrete Action Space Solution

The solution approach chosen to solve the discrete action space case was the widely used advanced policy gradient algorithm known as Proximal Policy Optimization (PPO). PPO is an algorithm that can handle larger state spaces in a computationally efficient manner, unlike other primitive policy optimization methods. The PPO algorithm is an actor-critic algorithm that uses actor and critic neural networks to learn an optimal policy for a given environment. The actor neural network is trained to output a probability distribution over all the possible discrete actions by maximizing the expected return. The critic neural network is trained to estimate the advantage function given the current state and/or action by minimizing the mean-squared error between the predicted value and the expected return. PPO has improved upon

vanilla policy optimization by clipping the object function to prevent large policy updates and using a ratio between the new and old policies to avoid large policy updates. These combined methods are known as the clipped surrogate objective function [5].

When training the actor/critic neural networks, the agent must interact with the environment and learn from the experience in order to maximize the agent's reward. The exploration/exploitation action selection process applied to gain the necessary experience was done by simply sampling from the actor neural network's probability distribution output based on the agent's current state.

The following pseudocode algorithm describes the implementation of the PPO algorithm used to find the optimal solution to the given race course environment.

1) Initialize hyperparameters
2) Randomly initialize critic neural network and actor neural network
3) For episode in $1 : M$ do
    - Initialize random Ornstein–Uhlenbeck noise process $\mathcal{N}$ for action exploration
    - Get initial observation state s_1
    - For episode step in $1 : T$ do
        - Select action by sampling the actor neural network's action probability distribution
        - Execute action a and observe reward r_t and next state s'
        - Store the current state transition T in buffer
        - Increment episode score by the reward
        - If the episode has come to completion, train the actor/critic networks for K epochs
            * Sample batches of current batch size from the buffer
            * Compute advantage estimates $\hat{A}_1, ..., \hat{A}_T$
            * Calculate loss L using the clipped surrogate objective function
            * Optimize both the actor and critic neural networks using the Adam optimizer
        - clear current episode's buffer

After training, the top 10 resulting policies were recorded and tested. The best policy was chosen based on the policy's ability to complete the race course and how quickly the policy led the car to complete the course. A detailed list of hyperparameters used in the training of this agent can be found in §8.

4.2. Continuous Action Space Solution

To address the challenge of continuous action spaces, the team utilized a DDPG RL algorithm and applied it to the learning environment. Unlike the more commonly used DQN algorithm for discrete action spaces, DDPG is well-suited to handle continuous action spaces. It employs an actor-critic network formulation, where the actor network learns the policy through the gradient of the Q-value with respect to the action, and the critic network learns the Q-value through Bellman backup. The team implemented several modifications to the DDPG algorithm to converge at a better solution, including the use of a replay buffer for sampling and training and having target actor and critic networks that are updated and trained to prevent Q-divergence. Ornstein-Uhlenbeck noise was used instead of an epsilon greedy process to introduce the exploration/exploitation tradeoff by adding time-correlated random noise to the network's selected actions.[8] The noise is characterized by the size of the random variations (exploration) and the mean-reverting force (exploitation). The following pseudocode algorithm describes the implementation of the DDPG RL agent [2].

1) Randomly initialize critic network $Q(s, a, |\theta^Q)$ and actor $\mu(s|\theta^\mu$ with weights θ^Q and θ^μ
2) Initialize target networks Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
3) Initialize replay buffer R
4) For episode in $1 : M$ do
    - Initialize random Ornstein–Uhlenbeck noise process $\mathcal{N}$ for action exploration
    - Get initial observation state s_1
    - For episode step in $1 : T$ do
        - Select Action $a = \mu(s|\theta^\mu) + \mathcal{N}$ according to current policy and exploration noise
        - Execute action a and observe reward r_t and next state s'
        - Store experience tuple (s, a, r, s') in R
        - Sample a random minibatch of N transitions (s, a, r, s') from R
        - Set $y_i = r_i + \gamma Q'(s'_i, \mu'(s_i|\theta^{\mu'})|\theta^{Q'})$
        - Update critic by minimizing the loss $L = \frac{1}{N}\Sigma_i(y_i - Q(s_i, a_i|\theta^Q))^2$
        - Update the actor policy using the sampled policy gradient $\nabla\theta_\mu J = \frac{1}{N}\sum_i \nabla a Q(s_i, a_i|\theta^Q)\nabla\theta_\mu\mu(s_i|\theta^\mu)$
        - Update the target networks $\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$ $\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$

In order to determine the optimal parameters for the DDPG agent, the team utilized Bayesian optimization to construct a probabilistic model of the objective function and select the hyperparamters that maximize the optimization function. A detailed list of hyperparameters used in the training of this agent can be found in §8.

4.3. Onboard Sensors Solution

For this problem, since the action space was once again discretized according to **(4)**, the team used the same PPO

algorithm and hyperparameters detailed in §3.2.2 to train the agent for this learning environment. In order to bridge the gap between the onboard sensors and the supervisor capabilities, visual cues (dark red rectangles) were established along the race course, mounted onto each corner of the race track as shown in **Fig. 1**. The visual cues aid in verifying the correct direction of travel as well as impact the reward as described in §3.4.3. After training, the top 10 resulting policies were recorded and tested. The best policy was chosen based on the policy's ability to complete the race course.

5. Results

Using the solution approaches discussed in §4, the team was able to successfully train the agent to navigate the race course for all three levels of success of the problem. The results of each solution space are discussed and compared in the subsequent sections. It should be noted that due to differences in the reward functions for each problem, the magnitude of the rewards obtained do not cross over between problems and should not be compared. The learned optimal policies for each of the three cases are demonstrated in the project repository which can be found at §8.

5.1. Discrete Action Space Results

The average reward over 20000 episodes is displayed in **Fig. 2**. The policy converged after 19500 episodes of training.

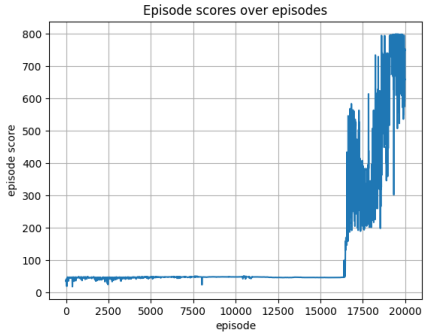

Figure 2. PPO Discrete Action Space Learning Curve

5.2. Continuous Action Space Results

The DDPG agent training on the continuous action space converged to a solution after 1200 episodes of training as is shown by the following average reward plot in **Fig. 3**.

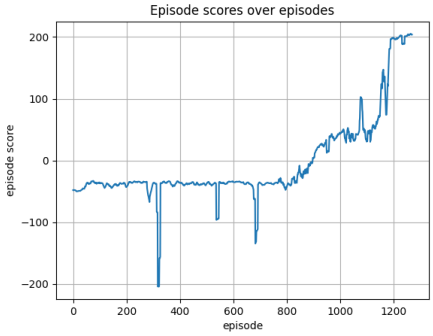

Figure 3. DDPG Continuous Action Space Learning Curve

5.3. Onboard Sensors Results

The average reward over 3800 episodes is displayed in **Fig. 4**. The policy converged after 3700 episodes of training and achieved the environment's solved condition.

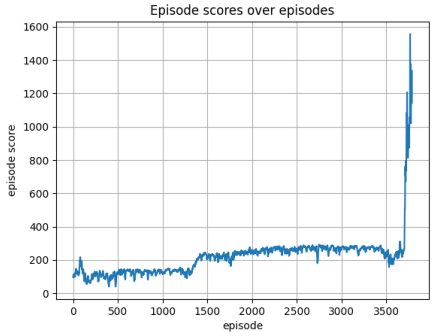

Figure 4. PPO Discrete Action Space Onboard Sensors Learning Curve

6. Conclusion and Future Work

In this project, the team considered three primary problem formulations covering both discrete and continuous action spaces as well as using Webots supervisor knowledge compared to only using onboard sensor knowledge. The team investigated and implemented PPO and DDPG RL algorithms in order to solve these problem formulations developing a greater understanding of the algorithms and the benefits/pitfalls of reinforcement learning. The team achieved success on all three subproblem levels outlined in **Table 1** finding policies that converged to the optimal policy allowing the car to navigate the course successfully.

Future work on this problem could include an investigation into the agents performance on a more complex track than the simple one outlined in this paper. Additionally, future work could investigate a dynamic environment with moving obstacles that would prove to be more complex than the environment used in this paper.

7. Contributions

Member	Contribution
Buencamino, Adam	Set up discrete action environment, PPO algorithm
Krebs, Chris	Research, testing, and validation
Miles-Colthup, Jake	Set up continuous action environment, DDPG algorithm

8. Release

The authors grant permission for this report to be posted publicly.

Acknowledgments

The authors would like to thank Professor Zachary Sunberg for his guidance and support throughout the entire process. His expertise in the field of reinforcement learning and Markov decision process problem formulation has been invaluable in shaping this project and its results.

The authors would like to acknowledge the support provided by the University of Colorado Boulder and the Ann and H.J. Smead Aerospace engineering department for the conduction of this project.

References

[1] J. Vogel, "Tech Explained: Ackermann Steering Geometry," *https://www.racecar-engineering.com/articles/tech-explained-ackermann-steering-geometry/*,

[2] T. P. Lillicrap *et al.*, *Continuous control with deep reinforcement learning*, 2019. arXiv: 1509 . 02971 [cs.LG].

[3] M. J. Kochenderfer, T. A. Wheeler, and K. H. Wray, "Algorithms for decision making," in The MIT Press, 2022.

[4] J. Schulman, S. Levine, P. Moritz, M. I. Jordan, and P. Abbeel, *Trust region policy optimization*, 2017. arXiv: 1502.05477 [cs.LG].

[5] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, *Proximal policy optimization algorithms*, 2017. arXiv: 1707.06347 [cs.LG].

[6] V. Mnih *et al.*, "Human-level control through deep reinforcement learning," *Nature 518*, pp. 529–533, 2015.

[7] B. R. Kiran *et al.*, "Deep Reinforcement Learning for Autonomous Driving: A Survey," *IEEE Transactions on Intelligent Transportation Systems*, pp. 1–18, 2021.

[8] B. Lehle and J. Peinke, "Analyzing a stochastic process driven by ornstein-uhlenbeck noise," *Physical Review E*, vol. 97, no. 1, Jan. 2018. DOI: 10.1103/physreve.97.012113. [Online]. Available: https://doi.org/10.11032Fphysreve.97.012113.

Appendix A: GitHub Repository

https://github.com/JakeMilesColthup/DMUFinalProject

Appendix B: Agent Hyperparameters

TABLE 2. PPO AGENT HYPERPARAMETERS

Hyperparameter	Value
Actor Learning Rate	0.001
Actor Network Neurons/Layer	32
Actor Network Layers	3
Critic Learning Rate	0.003
Critic Network Neurons/Layer	32
Critic Network Layers	3
Num Epochs	5
Max Gradient Norm	0.5
Clip	0.2
Gamma	0.99

TABLE 3. DDPG AGENT HYPERPARAMETERS

Hyperparameter	Value
Actor Learning Rate	0.00060
Actor Network Units	17
Actor Network Layers	3
Critic Learning Rate	0.00023
Critic Network Units	144
Critic Network Layers	3
Batch Size	96
Replay Buffer Size	1000000
Sigma	0.17
Theta	0.42
Tau	0.01
Gamma	0.99