Single-Agent Car Parking using Reinforcement Learning

Matthew Euliano, Curtis Manore, Andrew Mizell
Department of Electrical and Electrical Engineering, Northeastern University, Boston, MA, USA
euliano.m@northeastern.edu, manore.c@northeastern.edu, mizell.a@northeastern.edu

Abstract

Autonomous vehicles have been extensively studied and developed to improve road safety and user convenience. In this paper, we apply deep reinforcement learning algorithms to the problem of simulated car parking. The Unity3D engine is utilized for simulation and the Unity Machine Learning Agent (ML-Agent) toolkit is applied to compare the performance of Proximal Policy Optimization (PPO), Soft Actor-Critic (SAC), and Twin Delayed Deep Deterministic Policy Gradient (TD3) algorithms on the car parking task. The parking task is evaluated across three environments, consisting of fixed perpendicular parking, fixed parallel parking, and dynamic parallel parking. Algorithm performance is compared based on cumulative reward achieved by each algorithm over time in every environment. We demonstrate that the PPO algorithm, implemented with behavioral cloning, converges to the optimal policy with the shortest training time in both the static and dynamic parking environments. While SAC learned with behavioral cloning, it started to show unstable results after behavioral cloning was turned off. TD3 was able to perform in the static parallel parking environment, but it converged to a suboptimal policy, making it unsuitable for the other parking environments.

Introduction

Self-driving cars present a promising and active area of research, with numerous companies and universities working on fielding the first street-legal, fully autonomous vehicle. The problem of safely transporting passengers between two points presents a host of challenges, including navigation, lane monitoring, and object avoidance to name a few. One of the specific functions that autonomous vehicles must be able to perform is parking, including both pull-in parking and parallel parking.

In fact, there are several models of vehicles that already boast some form of autonomous parking capability, such as the BMW 7 and 8 series or the Cadillac XT4 and XT5 models (Nerad 2021). These cars implement forms of machine learning to allow the vehicle to interpret its environment and park appropriately given the information received from its numerous sensors. Before these capabilities can be deployed, countless tests must be conducted to ensure the safety and effectiveness of the guiding algorithms. One specific method for training these algorithms is reinforcement learning. However, despite these functionalities already being applied in limited capacities, there exists little research comparing the effectiveness of different reinforcement learning (RL) algorithms for single-agent car parking tasks.

Reinforcement Learning (RL) denotes a class of unsupervised machine learning algorithms, where the algorithm learns how to act in environments without human guidance or interaction. In RL methods, an agent interacts with its environment to update a state-action policy through trial and error that maximizes the reward received over time. If properly trained, the learned policy eventually enables the agent to rapidly make effective decisions, even in complex environments (Sutton and Barto 2018). There have been numerous studies demonstrating the efficacy of these learned policies, with RL agents matching or exceeding human performance across a wide range of environments and scenarios (Schulman et. al. 2017). Additionally, there are various levels of complexity within the RL domain, but the methods best suited for dynamic, continuous environments – such as the parallel parking task – are usually variations of the Policy Gradient approach (Peters 2010).

Recognizing the efficacy of these methods, we aim to formulate a generalized deep reinforcement learning approach for solving the parallel parking problem for autonomous vehicles in a simulated environment utilizing policy gradient methods. The single-agent policy gradient algorithms we utilize include Soft Actor-Critic (SAC), Twin Delayed Deep Deterministic Policy Gradient (TD3), and Proximal Policy Optimization (PPO). The algorithms are trained by running several environments simultaneously and their performance is evaluated based on the expected reward (cumulative reward achieved over time). Different variations of the algorithm are first applied to a static environment, where the goal parking space is fixed, to develop effective hyperparameters for learning while also exploring the impact of behavioral cloning. Then these tuned algorithms are evaluated on a randomized environment, where the parking space is moved in each episode, and a perpendicular, pull-in, parking environment, to evaluate the learning performance of each algorithm.

Background

Markov Decision Process

If an environment has the Markov property, then its one-step dynamics allow for the prediction of the next state and expected next reward given the next state and action. A RL task that satisfies the Markov property is called a Markov Decision Process (MDP), which serves as a foundational framework for modeling sequential decision making (Sutton and Barto 2018). The goal of an MDP is to attain a policy that maximizes the expected total reward signal within the environment.

An MDP can be described by a four-tuple of its one-step state and action sets, (S, A, T, R), where S is the possible set of states the agent may visit in the environment, A is the set of actions available to the agent in each state; T is the transition function, $T(s'|s,\mathrm{a})$, which describes the probability of the agent moving to the new state (s') given the state-action pair; and R is the reward function, $R(s,a)$, which maps state-action pairs to rewards. Typically, RL agents do not know the transition or reward functions, so they must be learned through trial and error over time (Attra 2021). Moreover, the MDP framework enables the learning of value functions by mapping states, $V(s)$, or state-action pairs, $Q(s,a)$, to the expected future rewards (assuming optimal action selection). These estimations can then be used according to the policy to guide action selection, which then cycles into value iteration or policy iteration for improving the algorithm's performance.

Q-Learning

Q-learning is a process of state-action value updating for estimating the optimal state-action values, $q^*(s,a)$, through incremental transition updates for each state-action pair visited within an episode. The update calculation takes the form of:

$$Q(s,a) = Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)] \qquad (1)$$

In this equation, $Q(s,a)$ represents the current value estimate for the state action pair, which updates based on its previous value, as well as the step size (α) and the *TD-error.* The step size controls the learning rate of the function, and the TD error represents the difference between the current and target value estimates. The target value is calculated from the immediate reward, r, as well as the discounted future greedy reward (Sutton and Barto 2018). This equation is utilized in RL algorithms to sample numerous trajectories through an environment, eventually allowing the estimated values to converge to optimal target values. While Q-learning methods are inherently biased based on the initial state-action value estimates, a sufficiently accurate approximation achieved through training is often enough to produce an optimal policy.

Policy Gradient Methods

An alternative approach to value function methods, such as Q-learning, are policy gradient methods. These techniques rely upon optimizing parametrized policies with respect to the expected return (long-term cumulative reward) by gradient descent. Instead of relying entirely on value functions, these methods instead learn action preferences and assign state-action selection probabilities based on those preferences.

Policy gradient methods are specifically useful for overcoming traditional reinforcement learning problems related to the intractability resulting from uncertain state information or the complexity that arises from continuous states and actions (Peters 2010). Unlike Q-learning methods, which require discretized action spaces, policy gradient methods can learn policies for continuous action spaces by estimating the mean and standard-deviation of some Gaussian distribution to then sample from for action selection (Attra 2021).

It is important to note that some policy gradient methods still use value functions for updating action preferences. We refer to these policy gradient algorithms as *Actor-Critic* methods – where *actor* refers to the policy function and the *critic* refers to the value function (Sutton and Barto 2018). This is directly relevant to the Deep Deterministic Policy Gradient method, as well as the SAC and TD3 methods we discuss later.

Simulator

RL algorithms are often trained within simulators due to their ability to rapidly create vast quantities of training data under realistic conditions without the real-world consequences (Tanner 2022). Understanding the complexity of autonomous parking tasks, as well as the physical and financial safety risks of testing vehicles, the research team has elected to utilize the Unity3D engine for simulation. The Unity3D engine is a general-purpose video game engine widely used for both 2D and 3D simulation work. When used in conjunction with the Unity Machine Learning Agents Toolkit (ML-Agents), this engine allows for easy implementation of RL algorithm training, especially for inherently supported PPO and SAC algorithms (Unity 2022). Moreover, Unity3D and the ML-Agents toolkit allows for interfacing with Python using socket programming to develop and train different algorithms, which is perfect for evaluating the non-inherent TD3 algorithm (Tanner 2022). TD3 was developed with RLLib – an open source library for reinforcement learning frameworks – to interface between the Unity environment and the algorithm itself.

Related Work

There have been several studies conducted on the application of RL methods to autonomous vehicle tasks, such as parking.

Deep Deterministic Policy Gradient

The DDPG method is an algorithm that uses the off-policy, actor-critic architecture. Specifically, it uses the Bellman equation and off-policy information to learn the Q-values and then uses the Q-values to learn the optimal policy (Sousa et al. 2022).

While not as widely used for parking tasks as PPO, some papers have demonstrated the application of DDPG successfully, such as Sousa et al. (2022) and Junzuo and Qiang (2021). These studies demonstrated that one strength DDPG may have over PPO for this task is its ability to adapt to dynamic environments. Due to its off-policy nature, if the parking space changed to a new location, DDPG may be able to learn the new target parking technique based on the previous behavior policy of the old parking space.

Recognizing that this method has been effective in the past, the research team was interested in exploring possible improved iterations of the algorithm. We found that the Twin Delayed DDPG method (Fujimoto, van Hoof, and Meger 2018) and the Soft Actor-Critic method (Haarnoja et. al. 2018) are indirect successors to DDPG, as both were developed around the same time based on DDPG. We also found that these methods are well suited for the continuous action space of the parallel parking task and can be implemented effectively in the Unity3D engine, so we chose to use these algorithms over DDPG.

Multi-Agent Car Parking

Another study conducted by Omar Tanner (2022) analyzed the application of RL methods to multi-agent car parking problems. The project implemented Deep Q-Learning (DQN) and PPO methods in several environments of varying complexity to "investigate how to effectively model cars in a car park as a multi-agent system." These environments included single agent with fixed goals, and multi-agent with both fixed and dynamic goals. Tanner also utilizes Unity3D for simulating these environments and implements the deep Q-learning utilizing a PyTorch plugin. The project ultimately finds that Q-learning has several limitations in the established environments, mainly with regards to obstacle avoidance behaviors. Tanner (2022) also explains how PPO successfully models the car agents for both the single and multi-agent scenarios, which supports its efficacy in application to this study.

These findings informed the environments and methodologies utilized for this environment, specifically the utilization of Unity, the implementation of external algorithms, and the application of PPO. However, we chose not to explore the multi-agent case or the application of DQN. The multi-agent case involves forms of cooperative and competitive learning between the different agents, whereas we are only interested in exploring the performance of policies generated for the single agent case. Additionally, we are interested in exploring the actor-critic methods instead of DQN as they are better suited for our MDP and provide a different form of variance for comparison with PPO.

Project Description

We describe the problem of learning a control policy for navigating a car agent to a designated parking spot using three different deep RL algorithms.

Markov Decision Process Formulation

To effectively apply, train, and evaluate each of the target deep RL algorithms, we frame the parallel parking problem as a finite-horizon, continuous action space MDP taking place in a deterministic environment.

The state space is described for each algorithm based on the following features:

$$S = \{(\Delta X, \Delta Y), V, \theta\} \tag{2}$$

where the state (S) is defined by the agent's position relative to the goal position in the horizontal (X) and vertical (Y) axis, as well as the agent's front-facing velocity (V) and steering angle (θ) with respect to the vertical axis. The initial state for each algorithm and environment training episode will consist of randomized X, Y, and θ values with no velocity (V=0). Terminal states will be defined by the set of coordinates within the environment-dependent goal parking space, as well as any collisions the agent may have with obstacles or environment boundaries.

The continuous actions taken at each state are defined by:

$$A = \{\pm V, \pm \theta\} \tag{3}$$

which correspond to an increase, no change, or a decrease in the velocity or steering angle values respectively. The change in velocity is applied indirectly via motor torque between ±1,000 Nm, whereas the steering angle is bounded between ±40 degrees. This action space results in a deterministic transfer function, T, calculated by each algorithm.

Finally, the reward function is defined as:

$$R = \begin{cases} +100 \mid s = s_{goal} \\ -10 \mid s = s_{term} \\ -5 \mid t = T \\ -0.05 \mid \text{otherwise} \end{cases} \tag{4}$$

where the agent receives a fixed large reward for successfully navigating to the intended parking space, negative rewards for crashing or reaching the termination time step (T), and a small negative reward at every time-step to encourage faster parking.

Environments

We explored three environmental variations – fixed goal perpendicular parking, fixed goal parallel parking, and dynamic goal parallel parking. To simplify the parking environment, we defined a rectangular drivable area surrounded by walls with a set of parking spots filled with static vehicles. On collision with either a vehicle or wall, the episode terminates with a negative reward described above. The empty parking space was defined by two box colliders in Unity separated by the length of the car agent. When in contact with both, and with a velocity close to zero, the agent was considered successfully parked and the episode is terminated with a positive reward.

Assumptions We established the different Unity environments with several assumptions that guided the construction and movement of the agent. All movement was considered deterministically in the horizontal plane, and there was no pitch, roll, or slip mechanics to account for in the agent's movement. Additionally, each episode in the environments were static as there were no dynamic environmental changes within each episode – even if the location of the goal space changed between episodes as with the dynamic parallel parking case.

Fixed Goal Perpendicular Parking In this environment, an empty perpendicular parking spot is placed in the center of the environment and the car agent randomly spawns in the space, as shown in Figure 1.

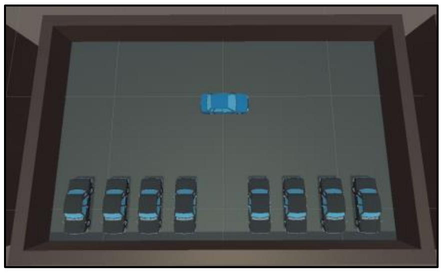


Figure 1: Perpendicular parking environment in Unity3D

Fixed Goal Parallel Parking The parallel parking environment has a fixed open parking spot in the center of the parked cars, with two cars on the left and right of the open spot. The open parking spot does not change when an episode terminates, and the agent must park in the spot successfully to receive a positive reward. The parallel parking environment is shown in Figure 2.

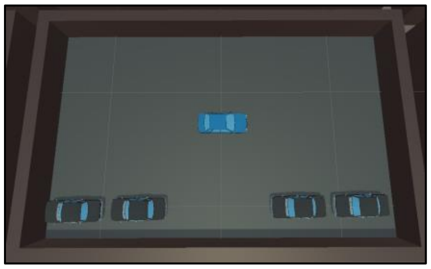


Figure 2: Parallel parking environment in Unity3D

Dynamic Goal Parallel Parking The dynamic parallel parking environment looks identical to the environment in Figure 2, except that the open spot is randomized between the center, right of center, and left of center spots. At the end of each episode, a new spot is chosen as the open spot to park in. The far right and left parking spots are not used as a potential open spot because they are too close to the environment boundary. This environment forces the agent to rely more heavily on its ray traces to better identify where the open spot is in the line of parked cars.

Algorithms

We describe the specific algorithms compared for this project and provide pseudocode for their implementations.

Twin Delayed DDPG The TD3 method builds on the DDPG algorithm for RL with a couple of modifications aimed at tackling overestimation bias with the value function. Specifically, TD3 utilizes clipped double Q-learning, delayed update of target and policy networks, and target policy smoothing (Fujimoto, van Hoof, and Meger 2018).

The clipped double Q-learning update is very similar to the original double Q-learning update found in Double-Deep Q-Networks (DDQN). In double Q-learning in an actor-critic architecture, an update is provided to both a pair of actors ($\pi_{\phi_1}, \pi_{\phi_2}$) and critics ($Q_{\theta'_1}, Q_{\theta'_2}$):

$$\begin{aligned} y_1 &= r + \gamma Q_{\theta'_2}(s', \pi_{\phi_1}(s')) \\ y_2 &= r + \gamma Q_{\theta'_1}(s', \pi_{\phi_2}(s')) \end{aligned} \tag{5}$$

Where π_{ϕ_1} is optimized with respect to $Q_{\theta'_1}$ and π_{ϕ_1} is optimized with respect to $Q_{\theta'_2}$. In regular double Q-learning, using two Q-values for this optimization should eliminate the overestimation of the first Q-value as the Q-value estimates are independent. However, in an actor-critic architecture, the critics are not entirely independent, resulting in some state-action values still having the overestimation of the true value. To solve this, the authors change the target update by putting an upper bound on the less biased Q-value by the biased Q-value estimate.

Algorithm 1: TD3 (Fujimoto et al. 2018)

```
Initialize critic networks $Q_{\theta_1}, Q_{\theta_2}$ and actor network $\pi_\phi$
with random parameters $\theta_1, \theta_2, \phi$
Initialize target networks $\theta'_1 \leftarrow \theta_1, \theta'_2 \leftarrow \theta_2, \phi' \leftarrow \phi$
Initialize replay buffer $B$
for $t = 1$ to $T$ do
  Select action with exploration noise $a \sim \pi_\phi(s) + \epsilon$,
  $\epsilon \sim N(0, \sigma)$ and observe reward $r$ and new state $s'$
  Store transition tuple $(s, a, r, s')$ in $B$
  Sample mini batch of $N$ transitions $(s, a, r, s')$ from $B$
  $\tilde{a} \leftarrow \pi_{\phi'}(s') + \epsilon, \epsilon \sim \text{clip}(N(0, \tilde{\sigma}), -c, c)$
  $y \leftarrow r + \gamma \min_{i=1,2} Q_{\theta'_i}(s', \tilde{a})$
  Update critics $\theta_i \leftarrow \text{argmin}_{\theta_i} N^{-1} \sum (y - Q_{\theta_i}(s, a))^2$
  if $t$ mod $d$ then
    Update $\phi$ by the deterministic policy gradient:
    $\nabla_\phi J(\phi) = N^{-1} \sum \nabla_a Q_{\theta_1}(s, a)|_{a=\pi_\phi(s)} \nabla_\phi \pi_\phi(s)$
    Update target networks:
    $\theta'_i \leftarrow \tau\theta_i + (1-\tau)\theta'_i$
    $\phi'_i \leftarrow \tau\phi_i + (1-\tau)\phi'_i$
  end if
end for
```

In practice, this is done by taking the minimum of the two state-action estimates in the target update:

$$y_1 = r + \gamma \min_{i=1,2} Q_{\theta'_1}(s', \pi_{\phi_1}(s')) \tag{6}$$

This double Q-value update does not have the overestimation problem and may introduce some underestimation instead, which is still more preferred than the overestimated case. The authors then add noise to the target update and "clip" the added noise so that it stays close to the update while helping to smooth the target policy and reduce variance. TD3 also delays the target network update so the TD error remains small, which is an improvement over the DDPG approach that does not have a delayed target network update.

Soft Actor-Critic The Soft Actor Critic method is an off-policy algorithm that attempts to find the optimal stochastic policy. While most RL algorithms want to maximize the expected sum of rewards, the Soft Actor-Critic (SAC) method is different as its goal is to maximize the expected entropy of a policy over the state marginal of the trajectory distribution, $\rho_\pi(s_t)$.

$$J(\pi) = \sum_{t=0}^{T} \mathbb{E}_{(s_t,a_t)\sim\rho_\pi}[r(s_t, a_t) + \alpha\mathcal{H}(\pi(\cdot|s_t))] \tag{7}$$

The above equation shows the objective $J(\pi)$ to maximize the entropy, where α is the temperature parameter that determines the relative importance of the entropy term, $\mathcal{H}$, against the reward, controlling the stochasticity of the optimal policy. $J(\pi)$ is used in the SAC algorithm to update the weights in a policy gradient manner.

Algorithm 2: SAC (Haarnoja et al. 2018)

```
Initialize parameter vectors $\psi, \bar{\psi}, \theta, \phi$
for each iteration do
  for each environment step do
    $a_t \sim \pi_\phi(a_t|s_t)$
    $s_{t+1} \sim p(s_{t+1}|s_t, a_t)$
    $D \leftarrow D \cup \{(s_t, a_t, r(s_t, a_t), s_{t+1})\}$
  end for
  for each gradient step do
    $\psi \leftarrow \psi - \lambda_V \hat{\nabla}_\psi J_V(\psi)$
    $\theta_i \leftarrow \theta_i - \lambda_Q \hat{\nabla}_{\theta_i} J_Q(\theta_i)$ for $i \in \{1,2\}$
    $\phi \leftarrow \phi - \lambda_\pi \hat{\nabla}_\phi J_\pi(\phi)$
    $\bar{\psi} \leftarrow \tau\psi + (1-\tau)\bar{\psi}$
  end for
end for
```

Using the expected entropy maximization approach leads to a few advantages such as wider exploration to help capture multiple potentially optimal policies, and improved learning speed overall. SAC is a common choice for many continuous control problems, which is why it is used in this experiment.

Proximal Policy Optimization PPO is a policy gradient reinforcement learning approach that strikes a balance between sampling the environment and optimizing an objective function through stochastic gradient descent (Schulman et al. 2017). PPO is an on-policy algorithm that has demonstrated success in the continuous control domain such as the 3D Humanoid environment. This is relevant to the parallel parking domain because of the velocity and angle control involved in steering the car to its parking spot.

PPO builds on Trust Region Policy Optimization (TRPO) methods. TRPO maximizes an objective function with a constraint on the size of the policy update. This constraint on the policy update size is determined by the KL divergence limit, which measures the distance between probability distributions. Finding the KL limit is very computationally expensive and needs to be approximated. Even still, TRPO is a computationally intensive algorithm, so PPO further approximates the KL constraint with clipping:

Algorithm 3: PPO (Schulman et al. 2017)

```
for iteration = 1, 2, … do
  for actor = 1, 2, … do
    Run policy $\pi_{\theta_{old}}$ in the environment for $T$ timesteps
    Compute advantage estimates $\hat{A}_1, \dots, \hat{A}_T$
  end for
  Optimize surrogate $L$ wrt $\theta$,
  with $K$ epochs and minibatch size $M \le NT$
  $\theta_{old} \leftarrow \theta$
end for
```

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t[\min(r_t(\theta)\hat{A}_t, \mathrm{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)] \quad (8)$$

where the loss is bounded (clipped) and the surrogate L is optimized.

Unity actively supports an open-source machine-learning toolkit with a focus on PPO and SAC (Soft Actor-Critic) algorithms, both being shown to perform effectively in eight different reinforcement-learning task environments (Juliani et al. 2018). With PPO as the default algorithm, successful implementations of PPO have been deployed on several relevant personal and academic projects such as racing games (Holubar and Wiering 2020).

Experiments

The goal of this project is to determine the best algorithm for autonomous parking—one that reaches the maximum reward the fastest and exhibits similar behavior to how a human would park a car.

The algorithms were trained on two desktop computers, both with Nvidia graphics processing units (GPUs) for using GPU accelerated tensor framework. Both computers had an Intel i7-12700K, 32 GB RAM, and an Nvidia RTX 3070 GPU.

All algorithms started training on the static parallel parking environment. The Unity simulator and ml-agents plugin allowed for parallel training environments, allowing for a setup of 25 car agents training in a parking environment, with multiple environments running at the same time. This parallelization greatly increased the training speeds of the simulations.

Both PPO and SAC can take advantage of Behavioral Cloning (BC) and Generative Adversarial Imitation Learning (GAIL). Both behavioral cloning and GAIL are imitation learning approaches where a human demonstration can be recorded prior to training to show what the right policy looks like. BC trains an agent's policy to exactly mimic the actions shown in the demonstration or set of demonstrations. GAIL uses a second neural network called the discriminator to distinguish whether an observation or action is from the demonstration or from the agent's movement. Then the discriminator provides a reward for how close the agent's movement is to the demonstration. This adversarial approach helps learn a policy that produces states and actions like the demonstration. BC and GAIL can work in conjunction with one another to help better performance. The ml-agents framework supports using BC and GAIL for the PPO and SAC algorithms. Without implementation of BC and GAIL, the agent failed to receive a positive reward after 10s of millions of timesteps due to the sparsity of rewards.

TD3 could not use GAIL or BC because TD3 is not a supported algorithm within the ml-agents framework. The TD3 framework was built using RLLib, which provides a Gym interface to interact with the Unity environment. The RLLib framework does not support imitation learning approaches for pre-training currently.

If an algorithm performed well on the static parallel parking environment, it would move to train on the dynamic parallel parking environment with the same hyperparameters. This is to help demonstrate how robust the algorithm is to a stochastic environment, especially as it must rely less on the demonstrations for imitation learning. Once trained on the dynamic environment, the algorithm will attempt to train on the perpendicular parking environment to show how generalizable the algorithm is with the same hyperparameters. Successful demonstration of parking in all three environments will determine the best reinforcement learning algorithm for the single-agent parking task at hand.

Hyperparameter Tuning

PPO hyperparameters were chosen by combining intuition from the original paper (Schulman et al. 2017) as well as the toy Unity environments developed using the ml-agents toolkit. After increasing the batch and buffer size, we found stable performance in all three environments.

TD3 hyperparameters were originally based off the DDPG parking parameters used in the DDPG autonomous parking research as discussed earlier (Junzuo and Qiang 2021). However, after initial poor performance with those parameters in TD3, some of the parameters from the original TD3 paper (Fujimoto, van Hoof, and Meger 2018) were used instead. These parameters led to more stable performance with TD3.

For tuning SAC, the ml-agents toolkit has many example hyperparameter configurations that are shown to work with their toy Unity environments (3DBall, Walker, Pyramid, etc.). These parameters looked quite different than the original SAC paper, so after some initial failures with adapting the example hyperparameter configurations, we resorted to many of the hyperparameters listed in the original SAC paper (Haarnoja et. al. 2018). All the hyperparameters for each algorithm can be found in Appendix A, and all the code utilized for this project can be found in the repository listed in the references.

PPO Static Parallel Parking Performance

PPO performed well when training on the static parallel parking environment as it converged to the optimum policy by 15 million timesteps. With the aid of GAIL and behavioral cloning, the car agent pulls forward past the parking spot, backs into the opening, and pulls forward in the parking spot to center itself in the goal parking location.

This demonstrates an optimal policy that models what a human

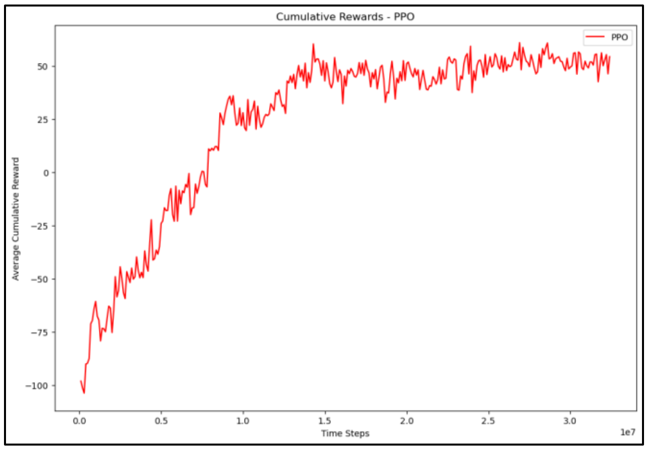


Figure 3. PPO cumulative rewards.

driver would do when parallel parking a car. The promising PPO performance in the static environment makes it a good candidate for the other two environments.

TD3 Static Parallel Parking Performance

The cumulative rewards for training TD3 in the static parallel parking environment are shown in Figure 4. While it does converge to a reward value at a timestep of 800,000, the converged policy is suboptimal. The average reward only reached 40, whereas the optimal policy reward for the environment is around 50.

When watching TD3 train on the parallel parking environment, the policy that TD3 converges to is one that puts the car agent in reverse and tries to arc the car into the parking spot as fast as it can. If the car spawns in a perfect placement where it can correctly arc itself into the parking spot, it will receive the optimum reward.

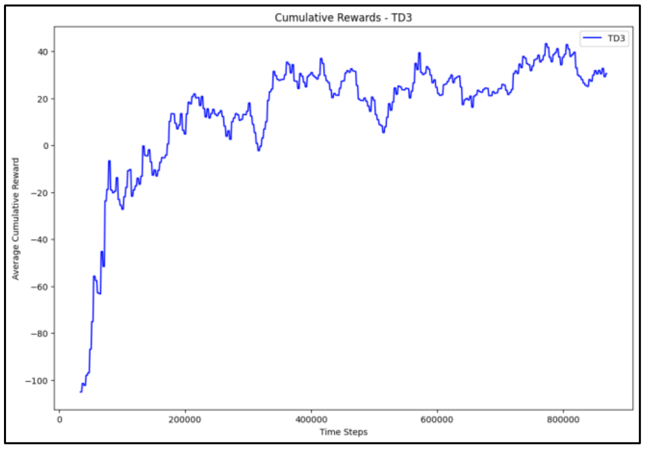


Figure 4. TD3 cumulative rewards.

However, if the car does not spawn in the center of the environment, it still tries to arc into the parking spot but either runs into obstacles in the environment or does not get a perfect alignment in the open parking spot.

This behavior was repeated past 800,000 timesteps, showing a convergence to a suboptimal policy. With TD3's suboptimal performance on the static parking test, it was not able to converge for the dynamic parking environment and was not used for comparison on the dynamic and perpendicular environments.

SAC Static Parallel Parking Performance

Soft Actor-Critic did not perform as well as TD3 or PPO. At the beginning of training, SAC seems to do well, but it is heavily reliant on the behavioral cloning that takes place for the first 750,000 timesteps. Once training surpasses 750,000 timesteps, behavioral cloning turns off and the SAC performance declines to the absolute negative reward. If the behavioral cloning timesteps are increased, the cumulative reward continues to increase and learn the BC behavior up until that timestep mark.

However, no matter how long BC is active for, performance declines afterwards as the agent cannot generalize the weights learned in the BC pre-training steps. Multiple SAC hyperparameter configurations and neural network structures were tried, but none led to stable performance in any environments after BC ends. Therefore, SAC was not used for further evaluation in the dynamic or perpendicular parking environments.

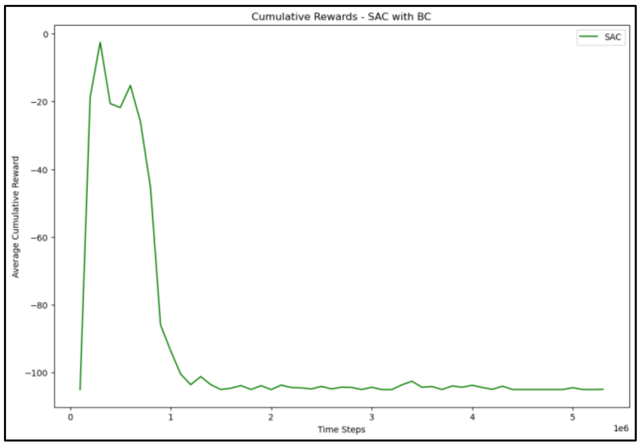


Figure 5. SAC cumulative rewards with BC.

PPO Cross-Environment Comparison

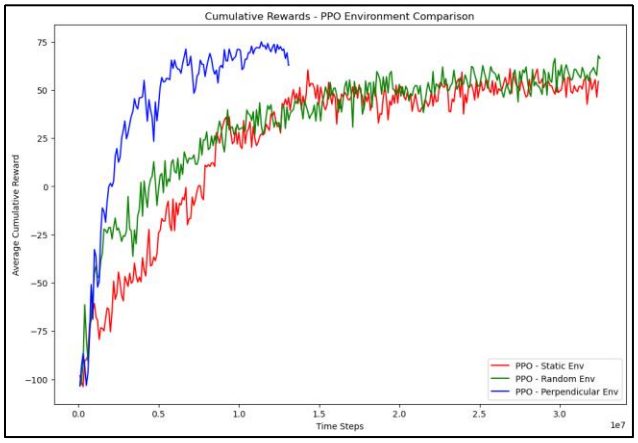


Figure 6. PPO comparison of cumulative rewards.

PPO was used in the dynamic (random) and perpendicular environments. When comparing the dynamic training to the static case, the dynamic training learns faster than the static training in the first 10 million timesteps—the cumulative reward is slightly higher, and the episode length is slightly lower for the dynamic environment. The agent experiences the environment variations in the dynamic case, causing it to learn what the goal reward is faster than the static case.

The perpendicular parking environment reaches the maximum positive reward faster than the parallel parking scenarios. Since the perpendicular environment involves a simpler movement than the parallel parking case, PPO learns the optimal policy faster. In all scenarios, the episode length decreased overtime and the cumulative reward converged, showing that PPO trained successfully in all parking environments to reach the optimal policy.

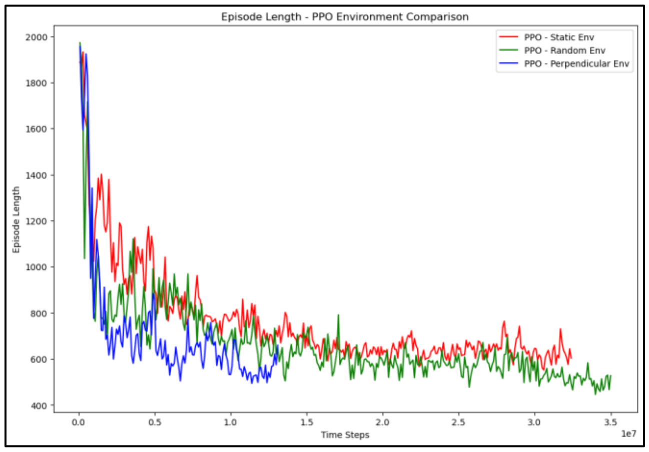


Figure 7. PPO comparison of episode lengths.

Conclusion

Rewards engineering is important to the success of reinforcement learning algorithms. In our case, when positive rewards were given to anything except a successful park, the agent learned to take advantage of those incremental rewards. For example, when a small positive reward was given to reaching one of the parking spot colliders, but not both, the agent learned to go in circles and repeatedly receive that reward until the end of the episode rather than parking. In the end, it was decided that a single positive reward for parking was necessary, even though the rewards were exceedingly sparse. The implementation of BC and GAIL helped in this case.

Deciding on the observations (state-space) available to the agent was important for computational efficiency and algorithm effectiveness. The state space was fed into the neural network, so the larger the state space, the more complex the neural net. An example of simplifying the state space was by not giving the agent a 4-element quaternion for its orientation, but a single value for Yaw rotation (because we assume no pitch or roll). Similarly, rather than feeding both agent position and goal position, we provided the difference in position.

Along these same lines, environmental simplicity is necessary to isolate problems as they arise. In initial training attempts, for example, the cumulative reward was inconsistent and appeared to unlearn policies. When the neural network was fed into the graphical Unity editor, we found that the advanced vehicle physics were causing the cars to flip over during high-speed turning. Reducing physics simulation complexity and creating a smaller drivable area restricted similar unwanted behavior.

In the end, we compiled an environment design, reward system, optimized state-space, and hyperparameter settings that consistently converged to a near-optimal policy using PPO, even with variation in environment and agent initialization.

Hyperparameter optimization was attempted using RLLib's Tune Library for the SAC and TD3 algorithms that underperformed. Tune offers Population Based Training (PBT), and Population Based Bandits (PB2) parameter sweeps to help optimize the hyperparameters. PB2 was implemented because of its claim to be a lightweight optimization tool. PB2 uses Bayesian optimization—using prior hyperparameter and performance information to decide which parameters to use next. Despite its claim to be a lightweight optimization tool, it used all 32GB of RAM on the test system, which crashed the simulation. In future work, PB2 can be used on a system with more memory, and if there is a memory leak in the PB2 algorithmic process, it can be identified for efficient parameter optimization.

Other future work includes expanding the sensory input used in the observation space. The car agents could have visual observations representing cameras, and those observations could be fed to a convolutional neural network to help train the agent to park visually instead of with ray tracing. While this could improve training performance, it would be more computationally expensive and require a more powerful computer to train on.

References

Attra, Gregory. 2021. Articulated Robot Control with Deep Reinforcement Learning.

Fujimoto, S.; van Hoof, H.; Meger, D. 2018. Addressing Function Approximation Error in Actor-Critic Methods. ArXiv:1802.09477.

Haarnoja, Tuomas; Zhou, Aurick; Abbeel, Pieter; and Levine, Sergey. 2018. Soft Actor Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor. arXiv:1801.01290.

Holubar, M. S., and Wiering, M. A. (2020). Continuous-action reinforcement learning for playing racing games: Comparing SPG to PPO. arXiv preprint arXiv:2001.05270.

J.D. Power. 2021. What Cars Park Themselves? These are the 10 Best Self-Parking Cars in 2021. https://www.jdpower.com/cars/shopping-guides/what-cars-park-themselves-these-are-the-10-best-self-parking-cars-in-2021. Accessed: 2022-10-25.

Juliani, A.; Berges, V. P.; Teng, E.; Cohen, A.; Harper, J.; Elion, C.; and Lange, D. (2018). Unity: A general platform for intelligent agents. arXiv preprint arXiv:1809.02627.

Junzuo, L.; and Qiang, L. 2021. An Automatic Parking Model Based on Deep Reinforcement Learning. *Journal of Physics: Conference Series*, 1883(1): 012111. Publisher: IOP Publishing.

Lillicrap, T.P.; Hunt, J.J.; Pritzel, A.; Heess, N.; Erez, T.; Tassa, Y.; Silver, D.; and Wierstra, D. 2019. Continuous Control with Deep Reinforcement Learning. arXiv:1509.02971.

OpenAI. 2022. Proximal Policy Optimization. https://openai.com/blog/openai-baselines-ppo/. Accessed: 2022-24-10.

OpenAI. 2018. Proximal Policy Optimization. https://spinningup.openai.com/en/latest/algorithms/ppo.html#pseudocode. Accessed: 2022-12-09.

OpenAI. 2018. Soft Actor-Critic. https://spinningup.openai.com/en/latest/algorithms/sac.html#pseudocode. Accessed: 2022-12-09.

OpenAI. 2018. Twin Delayed DDPG. https://spinningup.openai.com/en/latest/algorithms/td3.html#pseudocode. Accessed: 2022-12-09.

Peters, Jan. 2010. Policy gradient methods. *Scholarpedia* 5(11): 3698. Doi.org/10.4249/scholarpedia.3698.

Schulman, J.; Wolski, F.; Dhariwal, P.; Radford, A.; and Klimov, O. 2017. Proximal Policy Optimization Algorithms. arXiv:1707.06347.

Sousa, B.; Ribeiro, T.; Coelho, J.; Lopes, G.; and Ribeiro, A. F. 2022. Parallel, Angular and Perpendicular Parking for Self-Driving Cars using Deep Reinforcement Learning. In *2022 IEEE International Conference on Autonomous Robot Systems and Competitions (ICARSC)*, 40–46.

Sutton, R. S. and Barto, A. G. 2018. *Reinforcement Learning: An Introduction*. Boston: MIT Press.

Tanner, O. 2022. Multi-Agent Car Parking using Reinforcement Learning. arXiv:2206.13338.

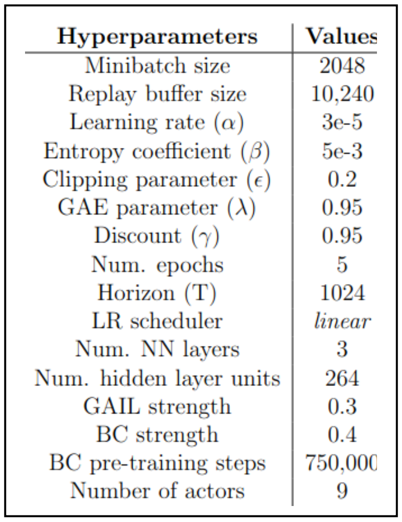
Unity. 2022. Unity Machine Learning Agents. https://unity.com/products/machine-learning-agents. Accessed: 2022-26-10.

Link to Repo

https://github.com/meuliano/RL_Parking

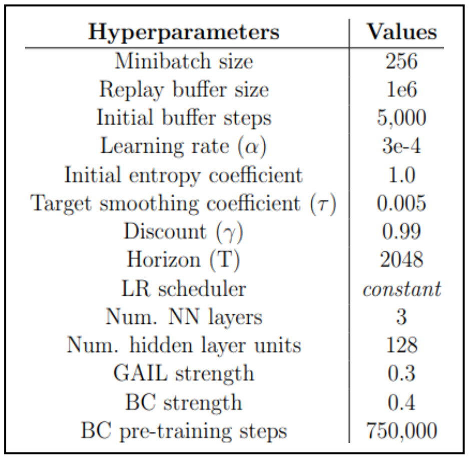
A. Algorithm Hyperparameters

Table 1. PPO Hyperparameters



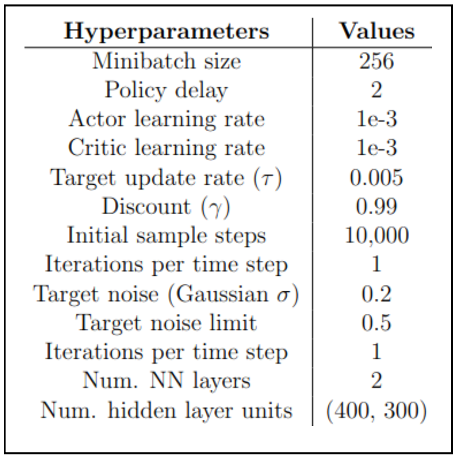
Hyperparameters	Values
Minibatch size	2048
Replay buffer size	10,240
Learning rate (α)	3e-5
Entropy coefficient (β)	5e-3
Clipping parameter (ϵ)	0.2
GAE parameter (λ)	0.95
Discount (γ)	0.95
Num. epochs	5
Horizon (T)	1024
LR scheduler	*linear*
Num. NN layers	3
Num. hidden layer units	264
GAIL strength	0.3
BC strength	0.4
BC pre-training steps	750,000
Number of actors	9

Table 2. SAC Hyperparameters



Hyperparameters	Values
Minibatch size	256
Replay buffer size	1e6
Initial buffer steps	5,000
Learning rate (α)	3e-4
Initial entropy coefficient	1.0
Target smoothing coefficient (τ)	0.005
Discount (γ)	0.99
Horizon (T)	2048
LR scheduler	*constant*
Num. NN layers	3
Num. hidden layer units	128
GAIL strength	0.3
BC strength	0.4
BC pre-training steps	750,000

Table 3. TD3 Hyperparameters



Hyperparameters	Values
Minibatch size	256
Policy delay	2
Actor learning rate	1e-3
Critic learning rate	1e-3
Target update rate (τ)	0.005
Discount (γ)	0.99
Initial sample steps	10,000
Iterations per time step	1
Target noise (Gaussian σ)	0.2
Target noise limit	0.5
Iterations per time step	1
Num. NN layers	2
Num. hidden layer units	(400, 300)