Comparative Analysis of DQN, DDQN, and DDPG for Reinforcement Learning in the Racecar Gym Environment

Nina Cvetkovska and Ognen Ivanov

Mentor: Sonja Gievska

Abstract

Our aim with this paper is to present a multi-agent reinforcement learning method in a racing environment where the race car’s motor and steering actions are controlled by one of three types of agents (DDPG, DDQN and DQN). LiDAR, position, velocity and acceleration of the vehicle are the sensory inputs that the environment uses to replicate realistic driving dynamics. The objective is for the agents to compete and traverse the environment effectively while adjusting and learning from their surroundings and interactions with other agents. It was discovered that training enhances each agent's performance, and depending on the agent's architecture, the effectiveness of each strategy varies. The difficulties in training multi-agent systems, including agent-to-agent communication and shared environment dynamics, are also covered in this study. Part of these findings can be used as a starting point for follow-up research in multi-agent collaboration and competition in racing environments.

Introduction

In the past few years significant progress has been made in the field of multi-agent reinforcement learning, particularly in challenging competitive contexts like racing or autonomous driving. However, training the agents to cohabit and learn in dynamic environments, where agent interactions are non-deterministic and highly complex, continues to be difficult. The primary goal is to implement and evaluate three distinct reinforcement learning algorithms for autonomous race car control in a multi-agent racing scenario. To improve their driving performance, minimize crashes, and maintain the racing trajectory, the agents must correctly manage their motor and steering actions in reaction to the environments' sensory inputs. The core driving factor behind this project is the creation of autonomous systems capable of making real-time decisions and continuously learn in dynamic, real-world environments. The project expands on previous research in multi-agent reinforcement learning and autonomous driving. This work contributes to the corpus of knowledge by providing a multi-agent system in which each agent's performance is determined by both its own actions and interactions with other agents in a competitive environment. Prior research has concentrated on single agent performance or basic settings. Furthermore, it analyses the performance of three reinforcement learning algorithms: DDPG, DDQN and DQN.

Related Research

Reinforcement learning in autonomous driving and multi-agent environments has been the research subject of the following studies:

[1] *Deep Reinforcement Learning for Autonomous Driving*: This paper analyses the use of DDPG to correctly navigate complicated state and action spaces in continuous domains. The Open Racing Car Simulator (TORCS) is used to train agents for autonomous driving tasks.

[2] *Deep Reinforcement Learning Applied to a Racing Game*: This study uses DDPG to train agents to negotiate tracks without traffic or time limits in the CannonBall OutRun game, showcasing the algorithm's efficacy in autonomous racing.

[3] *"Deep Reinforcement Learning for Multiagent Systems: A Review of Challenges, Solutions, and Applications"*: This review delves into the use of deep reinforcement learning in multi-agent systems, addressing issues like partial observability and non-stationarity while showcasing applications and solutions across a range of fields, including autonomous driving.

Agents and Methods

Setup

The racecar_gym simulation framework is used in this research to apply reinforcement learning (RL) to autonomous racing. It is developed on top of Gymnasium (formerly OpenAI Gym) and employs physics-based simulation to create realistic racing dynamics. The framework's support for continuous action spaces, multi-agent setups and track modification enables diverse control strategies. The "circle_cw" environment is used for training and testing, with agents moving clockwise around a circular track. Each agent drives a race car with motor and steering actuators, using LiDAR, location, velocity, and acceleration sensors for perception. While the race car's default colour is blue, it is customizable. The agent's incentives are based on their progress (laps and distance), with penalties for collisions and rewards for frame completion and track advancement. The reward is calculated by taking into account the difference in progress between frames. If the agent crosses the starting line incorrectly (in the wrong direction), the reward changes accordingly. The goal is to finish a lap in 30 seconds or fewer without getting penalized for accidents, so that agents can improve their racing performance in real-world scenarios. The document's table section offers a more thorough description of the environment.

Classes

1. **QNetwork: Function Approximation for Q-Learning**

The QNetwork is a deep neural network used as a function approximator for Q-values in reinforcement learning algorithms such as DQN, DDQN and DDPG. Function approximation employs neural networks to generalize over broad and/or continuous state spaces, in contrast to classical Q-learning, which saves Q-values for state-action pairings in a table. The purpose of the QNetwork is to associate a state with a collection of Q-values, which stand for the anticipated future reward for any action that might be taken at that state.

Architecture and Methodology

The QNetwork consists of fully connected layers with ReLU (Rectified Linear Unit) activations to add non-linearity and improve the model's capacity to represent complex state-action mappings. He initialization (He et al., 2015) is used to initialize weights and mitigate the risk of vanishing or exploding gradients. The core methods of this class include:

- Initialization (_init_): Sets up the weights and network layers.
- Weight Initialization (_init_ weights): This ensures that all linear layers have the correct gradient flow during training by implementing He initialization.
- Forward Pass (forward): Determines the associated Q-values for every operation by taking a state as input.

As the function approximator for predicting Q-values and aiding in decision-making, this design is a fundamental component of DQN, DDQN and DDPG.

2. DQN: Deep Q-Learning for Continuous Action Spaces

In this instance, the DQN class has been modified to operate with continuous actions, whereas it was originally created for reinforcement learning problems with discrete action spaces. To estimate the Q-values for every discrete action in a state, a Q-network is trained in typical DQN. The DQN method must, however, employ updated action selection strategies for continuous action spaces in order to support continuous action outputs while maintaining a Q-value approximation that is similar to a discrete action space.

Architecture and Methodology

Q-Network Architecture: To increase stability, the Q-network in DQN is trained using a target network and produces Q-values for a given state. The network is modified to forecast continuous values for the actions (such as steering and motor) in continuous action spaces.

Experience Replay: To deal with continuous actions in a consistent manner and prevent the instability frequently associated with continuous space approximations, the Replay Buffer retains previous experiences and samples batches during training.

Networks: Similar to the original DQN, there are two networks (one for assessment and one for action selection), which are updated on a regular basis to facilitate training for continuous action outputs.

Key Methods

- **__init__**: Sets up the optimizer, loss function, target Q-network, and Q-network for continuous action spaces. It configures the learning parameters (e.g., learning rate, batch size, discount factor) and the experience replay buffer.
- **preprocess_state**: Depending on the input type (e.g., dictionary, numpy array, or tensor), preprocess_state transforms the state into a flattened tensor in order to prepare it for input into the neural network. This guarantees that the state representation is appropriate for predictions of continuous actions.
- **update_memory**: This adds a transition to the replay buffer (state, action, reward, next_state, done). In order for the agent to learn from prior experiences in a consistent way, even in continuous action spaces, is crucial for the experience replay mechanism.
- **get_action**: Selects an action based on the current state and the epsilon-greedy policy, which balances exploration and exploitation. The output of continuous action spaces, such as motor and steering values, is a continuous action. To guarantee proper action ranges, Q-values are created for each action dimension and then clipped.
- **update_target_model**: Synchronizes the target network with the main model. This update is done periodically to improve training stability, especially when dealing with continuous actions, as this helps reduce instability in Q-value updates.
- **train**: Selects a batch from the replay buffer to carry out one training step. The target Q-network is used to update the Q-values for the continuous activities according to the temporal difference error. Faster training on continuous action problems is made possible by the method's efficient use of mixed precision training.

3. DDQN: Double Deep Q-Learning for Continuous Action Spaces

The DDQN class improves on DQN by using two separate Q-networks (one for action selection and one for value evaluation), ensuring more accurate Q-value estimations. This class is also adapted for continuous action spaces, where action outputs (like motor control and steering angle) need to be continuous rather than discrete.

Architecture and Methodology

Double Q-Learning Network: In DDQN, actions are chosen using the primary Q-network and evaluated using the target Q-network. Because it lessens the overestimation bias that can occur when working with continuous values in action selection, this design is especially advantageous for continuous action spaces.

Experience Replay: Like DQN, DDQN also uses experience replay, which allows the model to store and sample past experiences. By exposing the model to a variety of events, this helps avoid learning instability for continuous activities.

Networks: When working with continuous actions, DDQN offers more steady learning since the target network for both action selection and value estimate is updated on a regular basis.

Key Methods

- **__init__**: The primary distinction in initialization between DDQN and DQN is that DDQN employs two networks (one for action evaluation and one for action selection) for continuous action space issues.
- **train**: The key difference from DQN is that DDQN reduces overestimation bias by computing the target Q-values using two Q-networks. To increase training efficiency, the networks are updated using mixed precision and continuous action values are modified based on the temporal difference error.
- The remaining methods are identical to those used in the DQN Class.

4. DDPG: Deep Deterministic Policy Gradient for Continuous Action Spaces

The DDPG class is especially developed for reinforcement learning challenges that use continuous action spaces. It employs parts of Q-learning and policy gradient approaches to handle continuous action spaces in a consistent and efficient manner. The agent learns both a deterministic policy (actor) and a Q-function (critic), with the actor picking continuous actions (motor and steering) and the critic evaluating those actions based on projected future rewards. In the instance of DDPG, we employ a Q-network (critic) to estimate the Q-values for given state-action pairings and an actor network to generate continuous actions (motor and steering). Both networks are frequently updated with target networks to provide stability and effortless learning for continual activities.

Architecture and Methodology

Actor Network: The actor network in DDPG accepts state as input and produces continuous actions (motor and steering). The actor is responsible for learning a deterministic policy.

Critic Network: The critic network analyses the actor's actions based on the Q-values for the state-action pairings. The critic is trained on the Temporal Difference (TD) mistake.

Experience Replay: The same method for Replay Buffer, as the other agents, is used.

Target Networks: Both the actor and critic networks have target versions that are updated on a regular basis to ensure that the training process remains stable. This reduces divergence in Q-value estimations while training in continuous spaces.

Key Methods

- **__init__**: Initializes the actor and critic networks, the target networks, and the optimizers. It also sets the learning parameters like the learning rate, batch size, discount factor (gamma). The experience replay buffer is also initialized here.
- **preprocess_state, update_memory** and **get_action**: Are virtually identical across all classes.
- **update_target_model**: Synchronizes the actor and critic target networks with the primary models. Updates to the target models serve to smooth out the learning process and provide consistency in Q-value updates, particularly in continuous action spaces.
- **train**: The temporal difference error is used to update the critic network's Q-values, while the actor is updated to maximize the predicted Q-value. Mixed-precision training improves efficiency, especially when dealing with huge state spaces and continuous action outputs.

Parameters

All three agents (DQN, DDQN and DDPG) share key hyperparameters: a learning rate of 0.001, a discount factor of 0.95, and the use of a CUDA device. They all have a batch size of 4 and a memory capacity of 150,000. Adam is used as the optimizer, and MSE serves as the loss function. While DQN and DDQN both use QNetworks with 2 hidden layers, each having 64 units, for both the model and target, DDPG differentiates itself by using separate actor and critic networks, each with 2 hidden layers, the first with 256 units and the second with 128 units.

Tool and techniques

1. *HE Initialization*

Deep reinforcement learning relies heavily on neural network weight initialization for training stability and convergence speed. He initialization (He et al., 2015) is a popular strategy for initializing weights, which is especially useful for networks that utilize ReLU activation functions. This approach assigns weights to each layer using a normal distribution with a mean of 0 and a variance of $\frac{2}{n}$, where n is the number of incoming connections to the layer. In the context of reinforcement learning, the stability given by He initialization is critical for training large, complicated networks with sparse or noisy reward signals.

2. *Autocast for Mixed Precision Training*

Autocast for mixed precision training is a popular technique in current reinforcement learning algorithms for increasing training efficiency. Mixed precision training mixes 16-bit and 32-bit floating-point arithmetic, allowing models to use lower precision for specific computations, reducing memory use and speeding up training while maintaining model accuracy. The autocast functionality in libraries such as PyTorch automatically manages variable casting to lower precision when applicable, ensuring that the most computationally costly operations are still performed in higher precision (i.e., 32-bit) to guarantee model stability. The autocast approach allows for quicker training durations and lower memory usage, making it especially useful in contexts with limited computing resources (Micikevicius et al., 2018).

Training and Evaluation

Training

Dual-Phase Training

The agents were trained in two separate phases: solo and multi-agent training. During the solo training phase, each agent, DDPG, DDQN and DQN, received individual training across 250 episodes. During this phase, each agent was trained autonomously, and it learnt how to interact with the environment, maximize its rewards, and optimize its policy. Following the first phase, the agents received multi-agent training for either 250 or 750 episodes. This was done to see if the quantity of several training episodes had a significant impact on the overall outcomes. During this phase, the agents interacted with one another in a shared environment, enabling them to learn in a dynamic, competitive, and cooperative context. This phase enhanced the agents' decision-making and strategy, taking into account the activities of other agents in the environment.

Only Multi-Agent Training

During this training variation, all agents (A, B and C) were trained solely in a shared environment for 1000 episodes, interacting with one another. The agents are annotated as followed: A (DDPG), B (DDQN) and C (DQN). The agents' learning was influenced by their interactions with one another, which required them to modify their policies and methods to the competitive and collaborative dynamics of the environment. This arrangement enabled an in-depth assessment of the agents' capacity to learn and adapt to complicated circumstances involving several interacting entities. This helps us determine if the solo training phase is even necessary in the training process.

Evaluation

1. Training Analysis

During training, we evaluated agent performance based on rewards, progress, and collisions across multiple training variations.

Rewards and Progress for Dual-Phase Training in the Solo Training Phase:

- *Figure 1. (Rewards)* shows that while DDPG initially has the largest average payout, it soon diminishes and eventually falls below DQN's average. DDQN, on the other hand, has the most consistent reward accumulation.
- *Figure 2. (Progress)* illustrates that while all agents initially make significant progress gains, DDPG quickly falls behind both DDQN and DQN. While DDQN gradually declines, it finally falls below DQN, which shows the most consistent growth across training periods.
- *Figure 3. (Collisions)* highlights collision tendencies. DQN maintains a steady collision rate, whereas DDPG begins with few collisions but quickly increases, eventually surpassing both other agents. DDQN follows a similar pattern, progressively decreasing collisions by the 100th episode before slowly increasing.

Rewards and Progress for Dual-Phase Training in the Multi Training Phase:

- *Figure 4. (Rewards)* demonstrates that DQN has a consistent reward income, while DDPG and DDQN begin with large payouts, but fall below DQN's average by the conclusion of the training. More multi-episode training has no substantial effect on these tendencies, however slight oscillations may be seen.
- *Figure 5. (Progress)* indicates that while all agents improve, DDPG has the greatest immediate improvements. However, DQN gradually stabilizes and produces the highest average improvement, with clear peaks in later training sessions.
- *Figure 6. (Collisions)* shows that the number of collisions stays constant throughout training modifications. DQN maintains a consistent collision count, whereas DDPG and DDQN begin with fewer collisions and progressively increase. DDPG shows a higher growth, whereas DDQN climbs more slowly but eventually surpasses DQN in collision rate.
- *Figures 7.* and *Figure 8. (Ranks)* reveal that rankings remain largely unchanged across the different multi-episode phase training variations. DQN alternates between 1^{st} and 3^{rd} position, DDQN seldom takes 1^{st} place, and DDPG consistently ranks 1^{st}.

Rewards and Progress for Only Multi-Agent Training:

- *Figure 9. (Rewards)* shows that in lengthy training periods, DDQN achieves the best reward accumulation, while DDPG experiences mild fluctuations Unlike hybrid training, DQN does not maintain a consistent reward income and gradually falls, eventually dropping behind both agents.
- *Figure 10. (Progress)* highlights that progress patterns in only multi-agent training resemble dual-phase training but with more pronounced episode-to-episode differences. DQN exhibits dynamic variations, alternating between high and low progress, while both DDPG and DDQN start strong but steadily decrease.
- *Figure 11. (Collisions)* indicates that both DDPG and DDQN encounter collisions at first but rapidly drop to practically zero, whereas DQN retains a collision count comparable to dual-phase training.
- *Figure 12. (Ranks)* illustrates ranking trends. DQN typically ranks 1^{st}, although in hybrid training frequently rates 3^{rd}. DDQN is mostly 3^{rd}, a departure from its previous 2^{nd}-3^{rd} oscillation. DDPG has a higher 2^{nd} place ranking rather than continuously being 1^{st}. These findings show that while DQN improves from only multi-agent training, DDQN suffers, while DDPG maintains a mid-tier performance level.

2. Testing Analysis and Discussion

Testing findings shed light on the real-world generalization of trained agents. Ranking trends observed are as follows:

- *Figure 13.* **(Dual-Phase Training with 250 Multi Training episodes):** DDPG predominantly secures 2^{nd} place, DDQN alternates between 1^{st} and 2^{nd}, and DQN frequently ranks 3^{rd}.
- *Figure 14.* **(Dual-Phase Training with 750 Multi Training episodes):** A notable shift occurs, with DDPG dropping to 3^{rd} place more often, a striking contrast to its first dual-phase training results. Meanwhile, DDQN achieves a large number of 1^{st} place finishes, and DQN maintains a balance between 1^{st} and 2^{nd} place.

- *Figure 15.* **(Only Multi-Agent Training):** DDPG consistently ranks 2nd and 3rd , never taking 1st place. DDQN dominates 1st position, and DQN's results mirror those observed in the first dual-phase training.

The observed tendencies can be attributed to the agents' learning stability and adaptation to changing environments:

- **DDPG's Performance Drop:** Although it performed well in early training, its drop indicates that it overfits certain cases and suffers with generalization, resulting in inconsistent testing results.
- **DDQN's Consistency:** DDQN consistently improves and is the most trustworthy agent in long-term assessments. Its reward stability and reduced collision rates lead to its higher testing rankings.
- **DQN's Adaptability:** Despite discrepancies, DQN maintains a consistent progress trend and adapts well to competitive contexts, resulting in higher positions in testing.

Finally, our data indicates that DDQN provides the optimum combination of stability and reward optimization, whereas DDPG fluctuates significantly, and DQN is the most constant in progress and collision reduction. These findings emphasize the significance of balancing exploration and exploitation in multi-agent reinforcement learning, especially in competitive racing situations.

Conclusion

The purpose of this study was to compare the performance of three reinforcement learning agents, DDPG, DDQN and DQN, in a multi-agent racing scenario. Our findings indicate that each agent has distinct strengths and weaknesses, with DDQN demonstrating the most consistent performance across training sessions, DQN demonstrating stability in progress and collision avoidance, and DDPG excelling in early training phases but exhibiting significant fluctuations in performance. The training findings showed trade-offs between reward stability, progress consistency, and collision rates. While DDPG initially provided large rewards and quick growth, its performance degraded with time, resulting in more accidents. DDQN displayed robustness by keeping a somewhat constant trajectory despite slight variations, but DQN was the most consistent in terms of advancement and collision rates. During testing, we noticed changes in ranking patterns. DDPG, which had previously dominated in hybrid training assessments, suffered a drop in performance, finishing mainly second or third. DDQN frequently ranked first, demonstrating its capacity to adapt effectively to multi-agent interactions. DQN's scores fluctuated but improved when compared to training, indicating its capacity to effectively generalize beyond training settings. These findings highlight the complexities of multi-agent reinforcement learning, where performance is impacted by variables such as training stability, flexibility, and long-term learning capacity. Future research should look toward improving agent designs and training approaches to maximize performance in a variety of settings.

References

Deep Reinforcement Learning for Autonomous Driving
Kiran, B., & Alahi, A. (2018). Deep reinforcement learning for autonomous driving. *arXiv*. https://arxiv.org/abs/1811.11329

Deep Reinforcement Learning Applied to a Racing Game
Sriram, N. (2019). Deep reinforcement learning applied to a racing game. *Personal Project*. https://nsrishankar.github.io/files/docs/projects/AI_DDPG_Outrun.pdf

Zhang, W., Li, J., & Chen, T. (2021). Deep reinforcement learning for multi-agent systems: A review of challenges, solutions, and applications. *ResearchGate*. https://www.researchgate.net/publication/349232873_Deep_Reinforcement_Learning_for_Multiagent_Systems_A_Review_of_Challenges_Solutions_and_Applications

K. He, X. Zhang, S. Ren and J. Sun, "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification," *2015 IEEE International Conference on Computer Vision (ICCV)*, Santiago, Chile, 2015, pp. 1026-1034, doi: 10.1109/ICCV.2015.123.

Micikevicius, P., Narang, S., Alben, J., Diamos, G., Elsen, E., Garcia, D., Ginsburg, B., Houston, M., Kuchaiev, O., Venkatesh, G., & Wu, H. (*2017*). *Mixed precision training*. *arXiv*. https://doi.org/10.48550/arXiv.1710.03740

Tables

Table 1

Observations

Key	*Space*	*Defaults*	*Description*
pose	Box(6,)		Holds the position (x, y, z) and the orientation (roll, pitch, yaw) in that order.
velocity	Box(6,)		Holds the x, y and z components of the translational and rotational velocity.
acceleration	Box(6,)		Holds the x, y and z components of the translational and rotational acceleration.
lidar	Box(<scans>,)	scans: 1080	Lidar range scans.
rgb_camera	Box(<height>, <width>, 3)	height: 240, width: 320	RGB image of the front camera.

Note: Observations are obtained by (possibly noisy) sensors. Parameters for the sensors as well as the level of noise can be configured in the corresponding vehicle configuration. In the scenario specification, one can specify which of the available sensors should be used. The observation space is a dictionary where the names of the sensors are the keys which map to the actual measurements. Currently, five sensors are implemented: pose, velocity, acceleration, LiDAR and RGB Camera. Further, the observation space also includes the current simulation time.

Table 2

Actions

Key	*Space*	*Description*
motor	Box(low=-1, high=1, shape=(1,))	Throttle command. If negative, the car accelerates backwards.
speed	Box(low=-1, high=1, shape=(1,))	Normalized target speed.
steering	Box(low=-1, high=1, shape=(1,))	Normalized steering angle.

Note: The action space for a single agent is a defined by the actuators of the vehicle. By default, differential racecar defines two actuators: motor and steering. The action space is therefore a dictionary with keys, motor and steering. Alternatively, the agent can control the target speed, but it must be defined in the scenario specification. The action space of the car is normalized between -1 and 1.

Table 3

State

Key	*Type*	*Description*
wall_collision	bool	True if the vehicle collided with the wall.
opponent_collisions	List[str]	List of opponent id's which are involved in a collision with the agent.
pose	NDArray[6]	Ground truth pose of the vehicle (x, y, z, roll, pitch, yaw).
acceleration	NDArray[6]	Ground truth acceleration of the vehicle (x, y, z, roll, pitch, yaw).
velocity	NDArray[6]	Ground truth velocity of the vehicle (x, y, z, roll, pitch, yaw).
progress	float	Current progress in this lap. Interval: [0, 1]
time	float	Simulation time.
checkpoint	int	Tracks are subdivided into checkpoints to make sure agents are racing in clockwise direction. Starts at 0.
lap	int	Current lap.
rank	int	Current rank of the agent, based on lap and progress.
wrong_way	Bool	Indicates whether the agent goes in the right or wrong direction.
observations	Dict	The most recent observations of the agent.

Note: In addition to observations obtained by sensors, the environment passes back the true state of each vehicle in each step (the state is returned as the *info* dictionary). The state is a dictionary, where the keys are the ids of all agents.

Figures

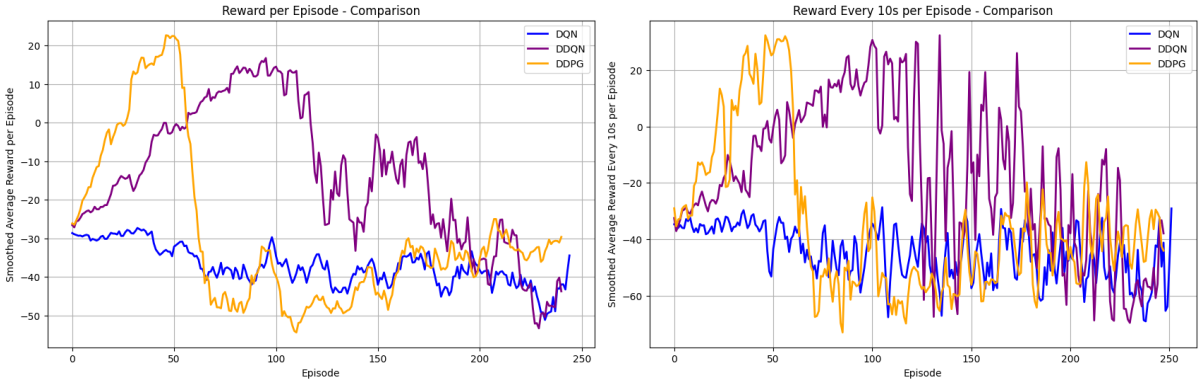

Figure 1. Rewards during the Solo Training Phase.

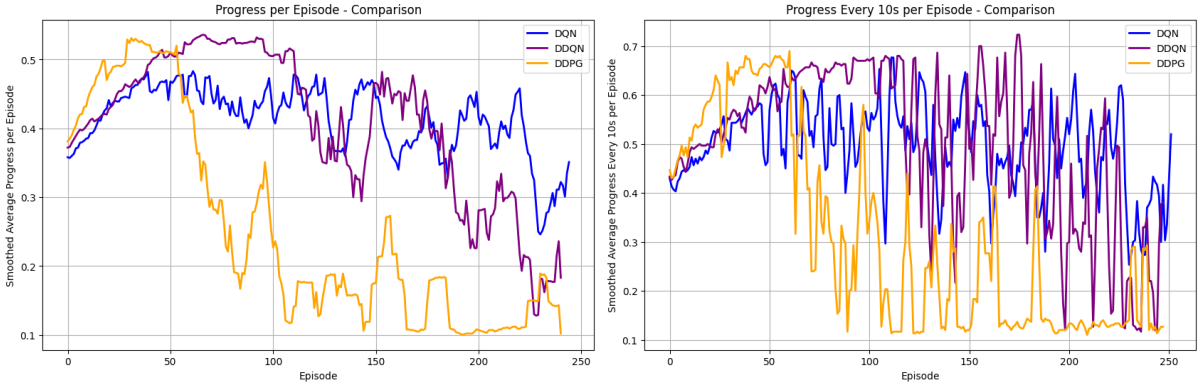

Figure 2. Progress during the Solo Training Phase.

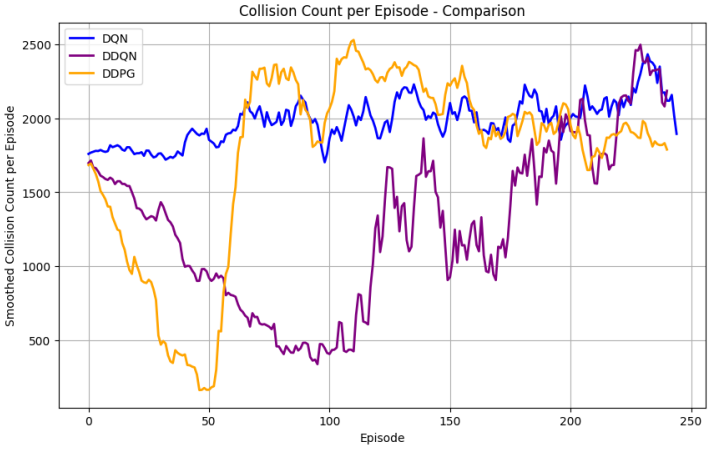

Figure 3. Collisions during the Solo Training Phase.

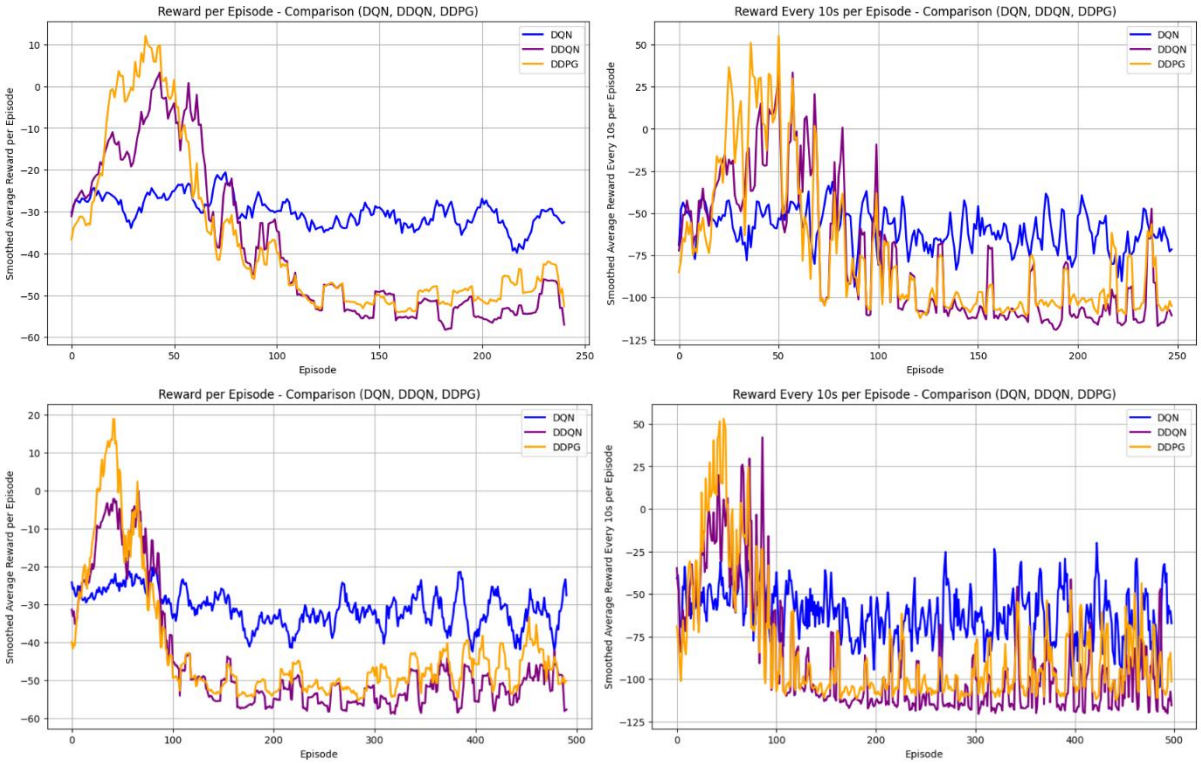

Figure 4. Top row includes rewards for Dual-Phase Training with 250 Multi Training episodes, and bottom row is Dual-Phase Training with 750 Multi Training episodes.

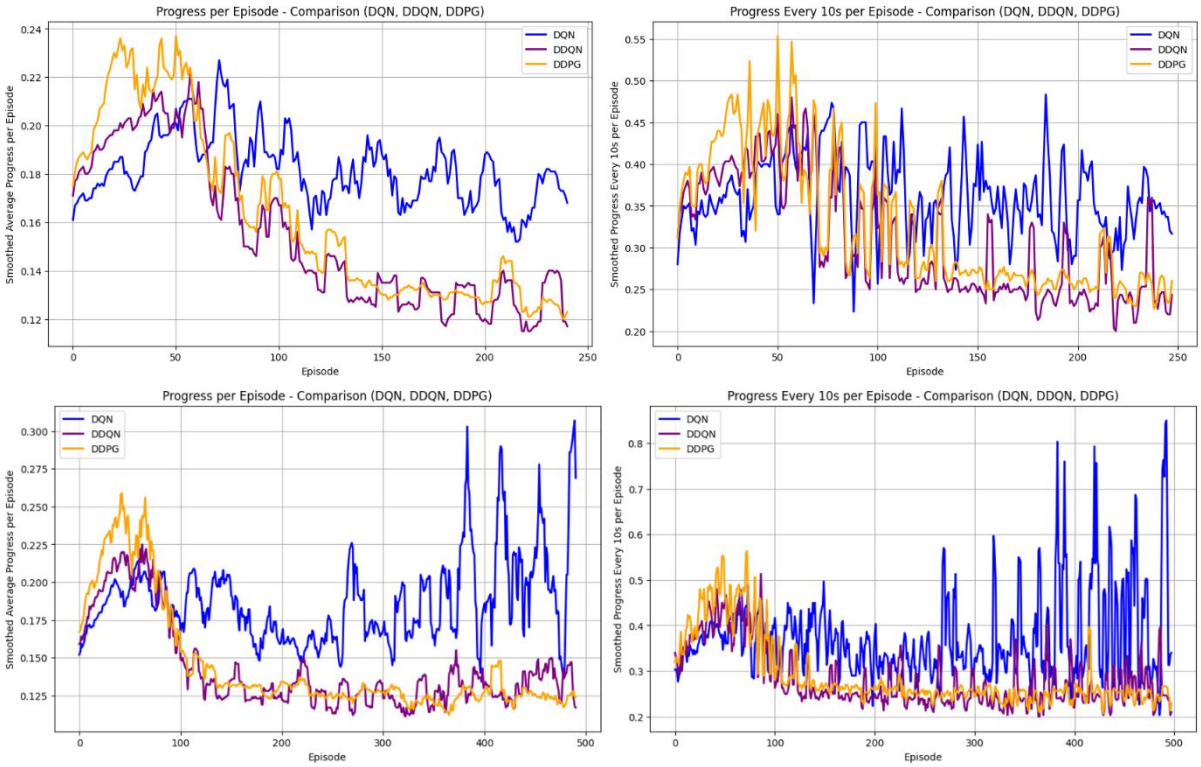

Figure 5. Top row includes progress for Dual-Phase Training with 250 Multi Training episodes, and bottom row is Dual-Phase Training with 750 Multi Training episodes.

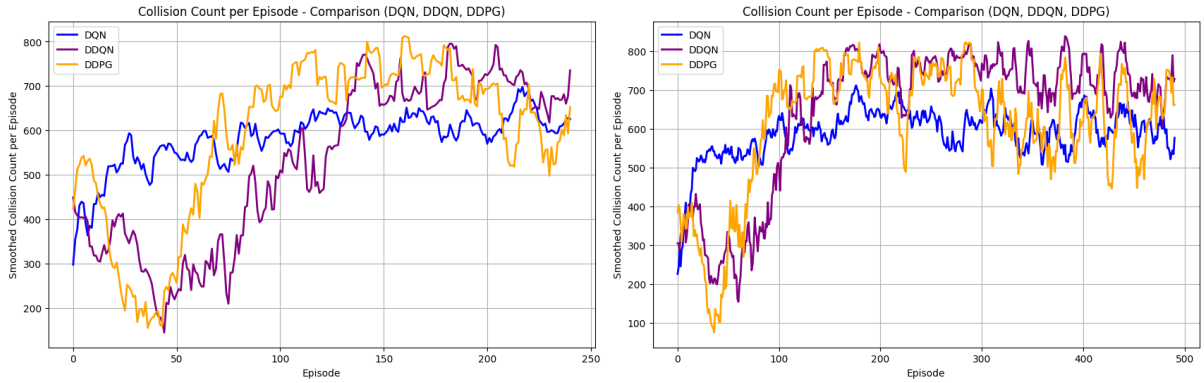

Figure 6. Left graph is the number of collisions for Dual-Phase Training with 250 Multi Training episodes, and right graph is Dual-Phase Training with 750 Multi Training episodes.

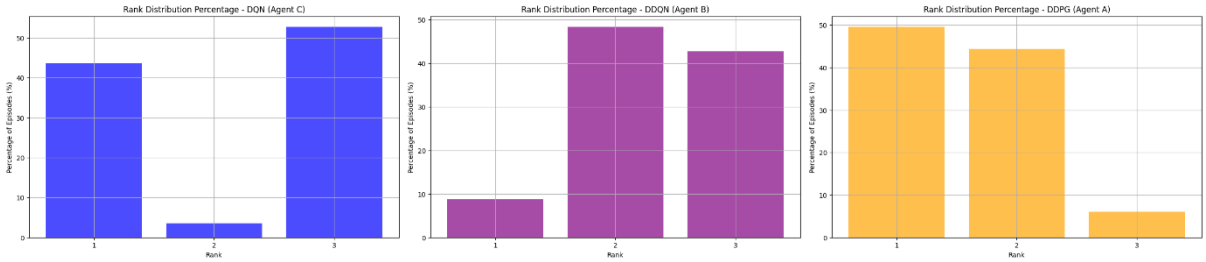

Figure 7. Rank counts for each episode in Dual-Phase Training with 250 Multi Training episodes.

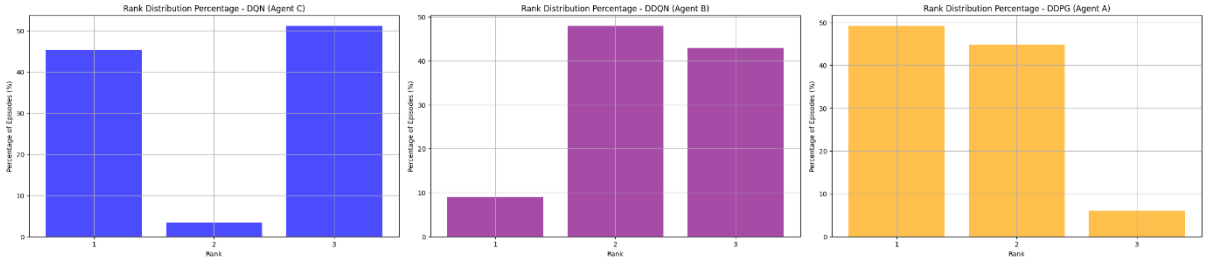

Figure 8. Rank counts for each episode in Dual-Phase Training with 750 Multi Training episodes.

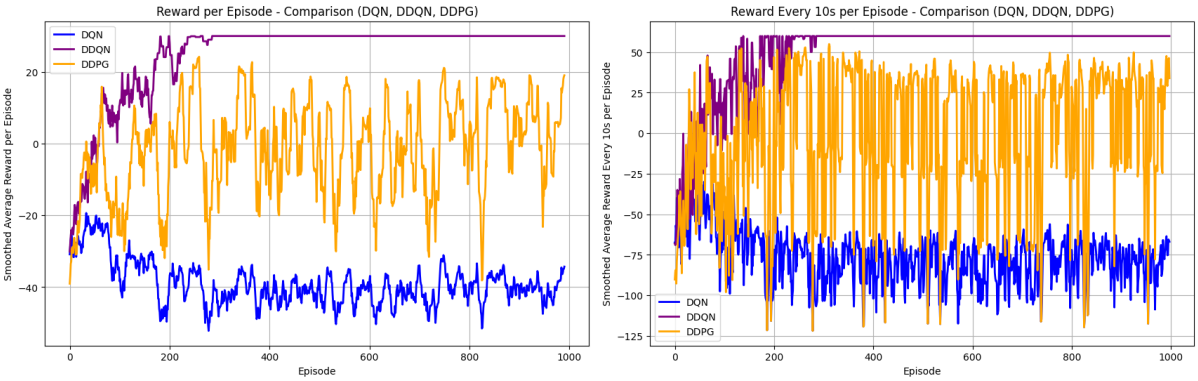

Figure 9. Rewards during the Only Multi-Agent Training.

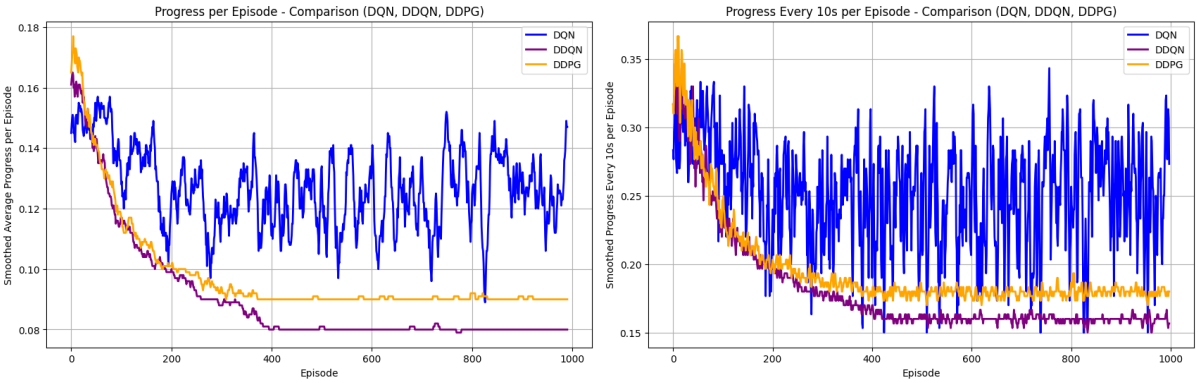

Figure 10. Progress during the Only Multi-Agent Training.

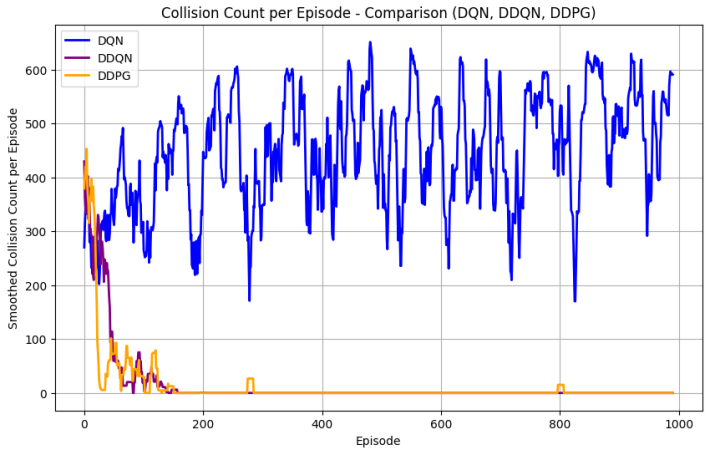

Figure 11. Collisions during the Only Multi-Agent Training.

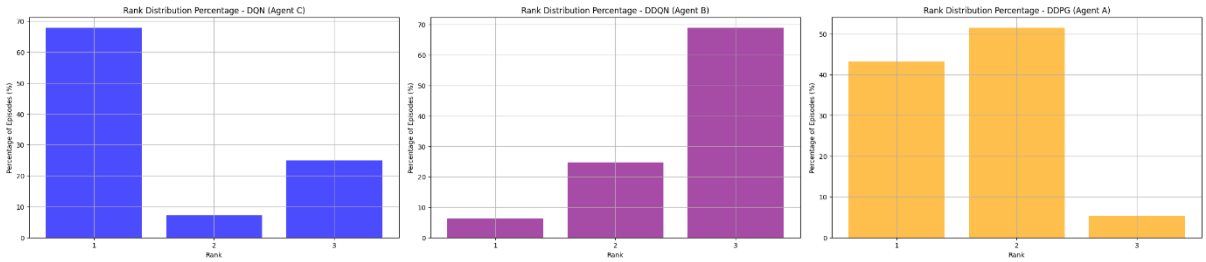

Figure 12. Rank counts for each episode in Only Multi-Agent Training.

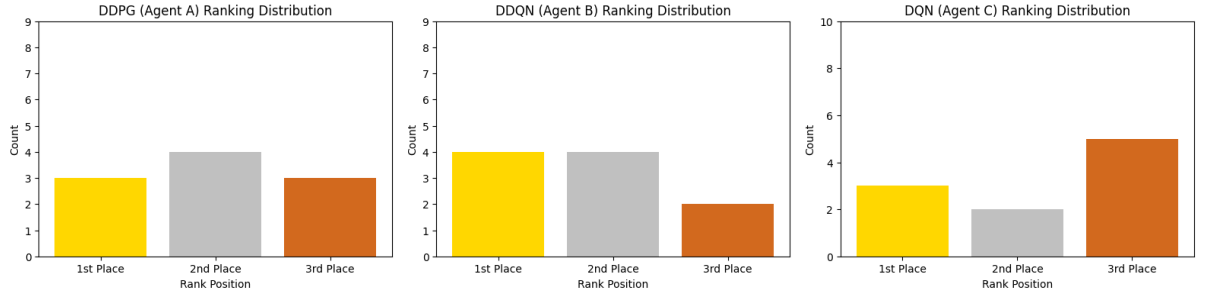

Figure 13. Ranks during Testing of Dual-Phase Training with 250 Multi Training episodes.

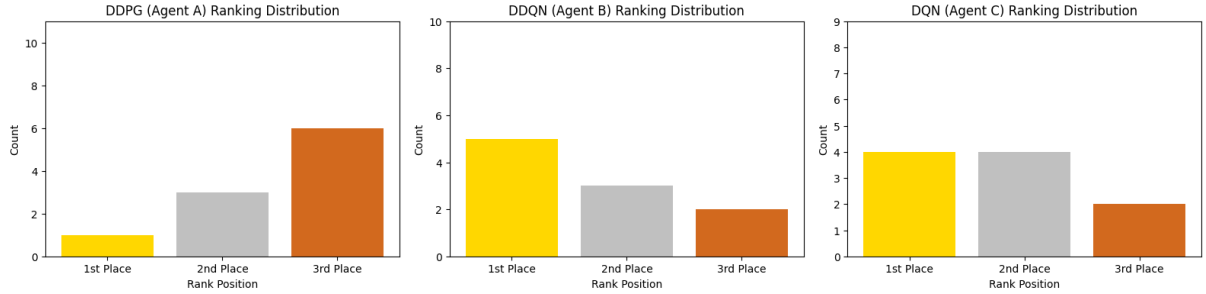

Figure 14. Ranks during Testing of Dual-Phase Training with 750 Multi Training episodes.

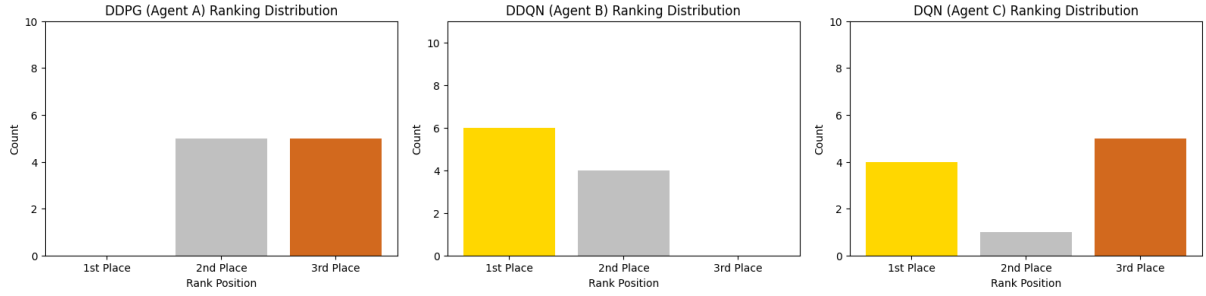

Figure 15. Ranks during Testing of Only Multi-Agent Training.