Imitation Learning for Car Racing Environment

Chao-Wu Chu (Kevin Chu), 85406312[*]
Department of Electrical and Computer Engineering
The University of British Columbia
Vancouver, BC V6T 1Z4
`cchu19@student.ubc.ca`

Abstract

With the fast development of machine learning, reinforcement learning has become a popular method to achieve learning goals through interacting with different environments. Recently, these systems have been applied to fields, such as robotics, large language models, and autonomous driving. However, since the environments of such applications are more complex and unstructured, a higher degree of adaptability and flexibility are required. The expected behaviors are difficult to define through reward functions, causing a challenging situation to specify the optimal rules and reward signal to manage the training. This is where imitation learning plays a vital role in these tasks. Learning from expert demonstrations that have been provided is a preferable solution, learning directly or assisting the original system.

This project aims to use PPO, behavioral cloning, and DAgger to train agents through a 2D car racing simulation environment provided by OpenAI, making the agent successfully navigate the vehicle throughout the race track. Through the experiments and results, we would like to discuss how imitation learning can be effective for this learning task through our implementation. We also want to discuss the moving patterns for continuous and discrete action spaces. We successfully trained the expert policy as a demonstration source to further train other policies with the imitation learning algorithms. All best models surpass 85% of the baseline performance, while only having the time cost lower than 10% of the baseline.

1 Introduction

1.1 Reinforcement Learning

In the field of machine learning, methods can generally be divided into three categories, which are supervised learning, unsupervised learning, and reinforcement learning. The requirement of labels is the difference between supervised and unsupervised learning, while the way of collecting data is how reinforcement learning is separated from the other two categories. Instead of relying on predefined raw data, reinforcement learning collects its data from the environment by its agent. This agent takes a sequence of actions to interact with the environment and gradually trains its controller module. In deep reinforcement learning, this module is usually a deep neural network, which is a common way of implementation nowadays. After its training, this network can be applied to the dynamic environment, controlling its agent to conduct expected actions and patterns for the task.

The goal is to let the agent learn how to act to maximize the expected long-term rewards with the environment state. The framework of reinforcement learning is based on the Markov Decision Process. This classic framework is to describe the decision-making process for the environments where the

[*]https://github.com/FalKon1256/UBC-EECE-571S

results are under partial control of the agent with randomness. The Markov Decision Process is based on the two assumptions. First, the environment needs to be Markovian, which means the state at the next time step is determined only by the current state, independent of the previous states. Second, the environment is fully observable, where the agent can observe all the information of the environment at any moment. The framework can be defined as a tuple of (S, A, T, R, γ), where S is a set of all possible states of the environment, A is a set of actions available to the agent, T defines the transition function which is the probability of moving from one state to the next state based on an action, R is the reward function as an estimation of the taken action, γ is the discount factor with a value between 0 and 1 determining the ratio of future rewards are considered.

To describe mathematically, each state and action is represented as a continuous form (a vector in R^n) or, in simple cases, a discrete value. The process starts from a sampled initial state s_0 for the first episode and continues with the iterated procedure. The agent picks an action a_n from A and observes the new state s_{n+1}, which is calculated from $T(s_n, a_n)$. The agent then receives a reward r_n from $R(s_n, a_n, s_{n+1})$. This is repeated until the termination conditions are reached, which can be defined as a terminal state or time limitation. The objective is maximizing the total reward $\sum_n \gamma^n r_n$, where $\gamma \in [0, 1]$.

1.2 Imitation Learning

Despite reinforcement learning being powerful enough to complete different tasks through an interactive agent, it still faces various challenges in many applications, such as robotics, autonomous driving, etc. These applications are involved in possible scenarios that are extremely complex, where the number of possible sequences of actions that an agent can take grows exponentially. This causes the reward to be difficult to define, having an inefficient learning process where too many cases are unknown. However, imitation learning provides a compelling solution to let the agent or policy learn directly or indirectly from demonstrations, performing desired behaviors or skills for a given task. This eliminates the need for explicit programming or task-specific reward functions.

Imitation learning refers to an agent acquiring behaviors from the demonstration of the expert as a teacher, which can be a trained policy or a human. The concept is similar to traditional supervised learning, while the examples are trajectories of the agent, formulated as pairs of states and actions, or in some cases there are rewards. The classic process of imitation learning starts from gathering demonstration data from the expert. These are usually encoded as state-action pairs, and the data are used to train the policy with supervised learning. However, some problems can occur when just using these demonstrations for training. First, the policy may not learn enough knowledge to navigate through the environment from direct mapping between state and action. This can happen because of errors during the collection of these demonstrations, insufficient number of demonstrations, or facing unknown observations when the policy starts to interact with the environment. To solve these problems, many improved imitation learning algorithms involve another step that requires the learning policy to perform the learned action or interact with the environment. Then the policy can be optimized according to its performance or current trajectory of the task for each update.

2 Related Work

In this section, we discuss some previous works of reinforcement learning and imitation learning algorithms, while we use some of them for our project.

For reinforcement learning at an early stage, research mainly focused on tabular or approximation-based algorithms [1], [2]. As the continuous study of this field, agents using TD error for value function update can be trained with on-policy methods, such as SARSA [3], or off-policy methods, such as Q-learning [4]. The conventional policy gradient reinforcement algorithm REINFORCE also came out during this time [5], using estimated cumulative rewards to update the policy. However, since the huge success of the Deep Q-network (DQN) in game-playing applications [6], deep learning techniques have become more popular in the field of reinforcement learning. Many works combined deep learning structures with reinforcement learning to solve complicated applications, such as robot control [7], autonomous driving [8], and games [9], etc. To tackle continuous action space, the deep deterministic policy gradient (DDPG) algorithm can learn deterministic policies through actor-critic architecture [10]. Trust region-based algorithms were also proposed during this time, such as TRPO [11] and its improved version PPO [12], which simplifies the TRPO algorithm using a truncated objective function. Other recent algorithms have also been proposed, such as offline reinforcement learning [13], using offline data collected in the past for the training process without interaction.

On the other hand, imitation algorithms can be implemented through different techniques or frameworks. Behavioral cloning solves the tasks through supervised learning [14], [15]. Although its simplicity is an advantage, this method faces the distribution shift problem, which is a huge weakness. To alleviate this problem, interactive imitation learning algorithms, such as dataset aggregation (DAgger), have been proposed [16]. Another way of implementing imitation learning is through the inverse reinforcement learning (IRL) algorithm [17]. This involves inferring the reward function with provided demonstrations from experts. The inferred reward function is used to train the policy through reinforcement learning. However, IRL algorithms often require executing reinforcement learning in the inner loop, causing high computational complexity as a drawback. As a promising solution, adversarial imitation learning emerges, introducing a two-player game dynamic of an agent and a discriminator to avoid solving the RL subproblem at each iteration. Generative Adversarial Imitation Learning (GAIL) is one of the methods that are popular in this category [18].

3 Method

In the method section, we introduce the reinforcement learning and imitation algorithms that we have used in this project. The actor-critic structure is also explained as our models are all based on this structure.

3.1 Actor-critic Structure

Many recent algorithms have started to adopt the actor-critic structure that combines both policy-based and value-based components. This method involves a policy as an actor and a value function as a critic with simultaneous learning. The actor generates the policies, selects actions, and interacts with the environment as in policy-based methods, while the critic evaluates the value function of the actor policy at each time step as in value-based methods.

For the process at each timestep, the actor first receives a state (or observation) and selects an action based on the current policy. Second, the environment returns the next state and the reward. The critic then evaluates the actor's policy and updates to approximate the value function. Finally, the actor uses the feedback from the critic to update its policy. Additionally, the critic can use different measures to evaluate the actor's policy, such as action-value function $Q(s, a)$, state-value function $V(s)$, or advantage function $A(s, a)$.

3.2 Proximal Policy Optimization (PPO)

PPO improves the REINFORCE algorithm by adopting an actor-critic structure while allowing some flexibility in data reusability from the previous policy. The main idea is to conduct gradient upgrade steps with importance sampling, which reuses its previous policy network to gain trajectory and update multiple times within a defined batch size, without making the policy deviate too much from the original policy behavior. With a replacement of KL divergence that is hard to estimate, a clipping mechanism is used as a constraint for the gradient update. This not only simplifies the loss function achieving comparable performance, but also stabilizes the network gradient. The clipping term is added to the action probabilities to prevent significant changes in the policy, where the loss function is defined as:

$$\mathcal{L}(\theta) = \mathbb{E}\left[\min(r_t(\theta)A_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)A_t)\right] \tag{1}$$

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} \tag{2}$$

where ϵ is the hyper-parameter, and $r_t(\theta)$ is the probability ratio of the action. In its actor-critic structure, the critic network is trained on a mean square error loss function to output appropriate values affecting the action selection. The advantage term is also used as a baseline to reduce the variance during the learning stage.

With these improvements, PPO is a widely adopted reinforcement learning algorithm due to its stability and trajectory reusing with a relatively simple framework. This method is also typically considered as an on-policy algorithm, but not completely because it still reuses the previous trajectory for a small number of training steps, which is an off-policy way of implementation in a local aspect.

3.3 Behavioral cloning (BC)

This is the most straightforward algorithm for imitation learning. The behavioral cloning algorithm is training a model or policy to replicate a behavior pattern from an expert through supervised learning. The training involves mapping states to actions with the provided demonstration, minimizing the difference between the predicted action and the expert action.

The whole process starts with collecting demonstrations from an expert, which can be either a trained policy or a human. The trajectories are state-action pairs (or state-state pairs in some variants of behavioral cloning), for example, the dataset may be the form of $D = \{(s_0, a_0), (s_1, a_1), \ldots, (s_n, a_n)\}$. Second, we enter the supervised learning phase, where a policy (or a neural network) is given the state from the dataset. The policy predicts the most likely action from its output distribution of the actions and then computes the loss function. Generally, we use a cross-entropy loss for discrete actions space, while computing a negative log-likelihood (NLL) loss (or mean-squared-error-like loss, MSE-like) for continuous action distribution. During training, the model learns a function that translates the current state into the corresponding expert action from the dataset. After training, the policy can use its learned function to generate actions when having actual interaction with the environment.

The huge advantage of behavioral cloning is that it does not need to interact with the environment during its training phase, while totally relying on the demonstrations. This makes it fast and easy to train without the need for any reward function, especially compared to reinforcement learning. However, it also faces a significant problem, which is the distribution shift. This occurs because the whole train is based on the states generated by the expert, but during actual testing, the agent encounters states that are induced by its own actions. In other words, this causes a shift in the observed state distribution between training and testing. The agent tends to make mistakes when running into any unseen states because of a lack of guidance on how to react or return to the states from the shown demonstrations. This eventually lets the agent drift and deviate from the demonstrated trajectory or behavior pattern each time when having out-of-distribution states. This disadvantage causes potential risks in many applications, especially when considering safety a priority, such as autonomous driving.

3.4 Dataset Aggregation (DAgger)

Since behavioral cloning suffers from the issue of distribution shift, many research works tried to solve this problem, and interactive imitation learning is one of the popular areas. Dataset Aggregation is an improved algorithm that falls into this category. This method reduces the mismatch between training and testing by collecting expert feedback at each round. An important assumption of this method is that the agent should be able to consult an online expert during training.

The whole process starts with an empty dataset $D = \emptyset$ and an initialized policy (or a pre-trained policy). The current policy interacts with the environment and collects the new states that it encounters, adding them to the dataset, where the dataset becomes $D = \{s_0, s_1, , \ldots, s_n\}$. Next, the expert policy labels the collected states based on its decision of the output action and combines them with the corresponding states in the dataset. The dataset becomes $D = \{(s_0, a_0^*), (s_1, a_1^*), \ldots, (s_n, a_n^*)\}$, where a^* represents the action suggested by the expert policy. The current policy is then trained with this aggregated dataset D, including both new and old data labeled by the expert. Finally, the current policy finishes its training after certain training rounds.

Huge advantages are gained with this algorithm compared to behavioral cloning. Dataset Aggregation might make mistakes during the early stage of training, but unlike behavioral cloning, this method has an expert policy to query during the whole training stage. In other words, even if mistakes were made in the previous trajectory, this method can still turn the collected data into useful information for the next training round through the expert policy. This makes the final trained policy far more robust than simply using behavioral cloning. However, some disadvantages still exist. Since the agent needs to perform interactions with the environment during each training round, this algorithm requires a higher computational cost than behavioral cloning. Another drawback is that dataset aggregation

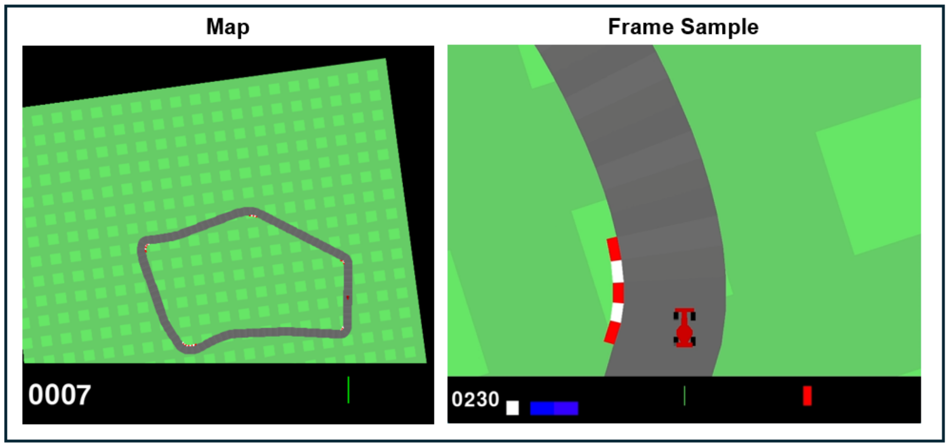

Figure 1: This figure shows the car racing environment, The left part is the whole picture of a random race track, while the right part is the frame sample when the player controls the vehicle.

needs access to the expert at all times, on the other hand, behavioral cloning only asks for the expert once for data collection before its training.

4 Experiments

In this section, we introduce our experiment settings in detail. Overall, all our experiments are in the "Car-Racing" environment developed by OpenAI as a car-racing-simulation environment. The whole process can be divided into two phases, the reinforcement learning phase and the imitation learning phase. First, in the reinforcement learning part, we adopt the PPO algorithm to train an expert policy. After training, we pick the best policy as our source of demonstrations. For the imitation learning phase, we use the vanilla BC and DAgger to train our models. Supervision signals are from the generated demonstration, which is provided by the selected expert policy from the previous stage. We conduct all experiments with continuous and discrete action space settings. Other testing experiments are also implemented, such as the generalization test and action distribution test. The setting details are explained in the following parts.

4.1 Problem Description

The Car-Racing environment has a car agent and a random race track map, shown as Fig. 1. Some indicators are shown at the bottom of each frame. The true speed, four ABS sensors, steering wheel position, and gyroscope are displayed from left to right. This environment is formulated as the Markov Decision Process (MDP) model with the state, action, transition, and reward information below:

Observation Space: The observation (or state) is a top-down 96 x 96 pixel RGB image of the vehicle, race track, and background information. The shape of each state is 96 x 96 x 3. The car starts at rest in the center of the road for every episode.

Action Space: For continuous case, there are 3 actions, which are steering (-1 is full left, +1 is full right), gas (any value between 0 and +1), and breaking (any value between 0 and +1). For discrete cases, there are 5 actions, do nothing, steer left, steer right, gas, brake.

Transition: The transitions from state to state are controlled by simulation physics and the dynamics of the vehicle. The wheel angle, gas, and break value can be changed according to how the player controls the car. The simulation physics computes and re-renders the frame for the next update. For each simulator computation, a new 96 x 96 RGB image is generated as a new state, reflecting the updated position and display of the vehicle and track.

Sample Trajectory: The trajectory can be expressed as $(s_0, a_0, s_1, a_1, \ldots, s_n)$, starting from state s_0 to s_n. For example, the initial state s_0 is an image of the vehicle in the center of the road. a_0 can speed up, so the next state s_1 may have the vehicle moving forward with the race track around.

Reward: The reward function is provided by the environment. The agent gains a positive reward of +1000/N for every visited track tile, where N is the total number of tiles in the race track. On the other hand, the agent receives a negative reward of -0.1 for every frame. If the car goes outside the playfield map, which is far off the track, the agent also receives a -100 reward (the episode ends immediately). To gain high scores, this setting of rewards encourages the agent to stay on the race track and to use an efficient driving style to visit as many tiles as possible.

Termination Condition: There are three cases for each episode to end. The first case is when all the track tiles are visited. The second case is when the agent has taken up to 1000 steps since it is a maximum for each episode in our setting. The last case is when the car goes outside the playfield map, the episode will end.

4.2 Policy Network

In the network setup, all our policy networks are based on actor-critic architectures. We use an MLP-structured and CNN-structured policy network for our experiments. The actor and critic network shares the same MLP or CNN backbone, connected with two separate linear heads for action or value output.

For the MLP actor-critic network, the feature extractor first flattens the input and then sends it to the actor-critic network, which is the MLP extractor above. The MLP extractor is a lightweight fully connected network with two hidden layers each consisting of 32 units and ReLU activations. On the other hand, the CNN actor-critic network, the CNN is directly used as the feature extractor. The structure contains three convolutional layers (Conv2d(3, 32, kernel=8, stride=4), Conv2d(32, 64, kernel=4, stride=2), and Conv2d(64, 64, kernel=3, stride=1)) with each a ReLU layer. A linear layer is then to flatten the vector at the end. The output from CNN is then passed to a fully connected layer with 512 units in the output layer, connecting with separate linear heads.

For both structures, the shared network outputs separate latent vectors to heads of the action net and value net, which are respectively linear layers mapping the latent vectors to the action space and scalar estimate value.

4.3 Obtain Expert Policy

This is the reinforcement learning phase, while we use the PPO algorithm to train the policy. We then pick the best policy with the highest average reward as our expert policy for both continuous and discrete cases. For the training hyperparameters, the learning rate is fixed to 0.0003, the batch size is 64, and the step number is 2048 (run for each environment per update). We save the checkpoints when reaching certain training steps, and then conduct the evaluation 20 times for each checkpoint.

4.4 Vanilla BC and DAgger Training

This is the imitation learning phase, where we adopt the vanilla BC and the DAgger algorithms to conduct supervised learning. Since we get the expert policy from the previous stage, we use it as a source of demonstrations in this part. For the loss function, the log probability represents the probability of whether the policy's predicted action matches the expert action based on an observation. As a result, the training goal is to minimize the negative log-likelihood, which is the main core concept of loss for Vanilla BC and DAgger. The whole loss function can be represented as:

$$\mathcal{L}_{total} = \mathcal{L}_{NLL} + \mathcal{L}_{Entropy} + \mathcal{L}_{L2} \tag{3}$$

where $\mathcal{L}_{NLL}$ is the NLL loss for action matching between the current and expert policy, $\mathcal{L}_{Entropy}$ is the entropy loss to encourage exploring, and $\mathcal{L}_{L2}$ is the L2 loss for regularization to prevent overfitting. For continuous cases, the action distribution is defined as a Gaussian Distribution, where the NLL loss is computed as an MSE-like loss. As the training goes on, the most likely action sampled from the policy becomes closer to the mean of the distribution, which is the action made from the expert policy. On the other hand, for the discrete cases, the distribution is defined as a Categorical Distribution, and the NLL loss is computed as a Cross-Entropy loss. It maximizes the Softmax probability of the expert policy's action. We also use actor-critic networks based on both CNN and simple MLP structure for the training.

For demonstration generation, vanilla BC and DAgger are different during their process. For vanilla BC, the expert policy generates all expert trajectories as demonstrations before the training starts. We experimented with different sizes of the demonstration dataset. The number of training epochs is fixed to 50 for the vanilla BC training, which is the number of complete passes made through expert data before ending training.

For DAgger, we do not provide demonstrations before the training. The initialized policy directly interacts with the environment to gain the observations, then the expert policy helps label these observations with its actions of highest probabilities based on each state. The next round starts again with an updated policy and continues until the training is finished. We train the policy with different numbers of total steps for DAgger training.

All vanilla BC and DAgger training uses the same checkpoint as expert policy (but different policy between continuous and discrete scenarios). The batch size is fixed to 32, while we record 500 batches as 1 step.

4.5 Other Experiments

We also conduct the generalization test and action distribution test. For the generalization test, we set the seed number to 1000, which is different from the original value of 0 that we used in previous sections. We evaluate how well the policy performs after vanilla BC or DAgger training, compared to the performance of the same baseline (the best expert policy trained by PPO). We select the best policy for each algorithm and compare the performance gain and time cost. For the action distribution test, we use the PPO expert policy, vanilla BC policy, and the DAgger policy to run 5 episodes in the environment (seed also set to 1000). We get the action distribution of both continuous and discrete action space to compare the difference between using different algorithms.

4.6 Hardware Settings

All experiments are implemented under the same hardware conditions to get fair results. We use the CPU of AMD Ryzen 7 7800X3D (8-core processor) and the GPU of NVIDIA GeForce RTX 4070 (12g VRAM).

5 Results

In this section, we show our experiment results in five parts, including the PPO training, BC Training, DAgger Training, generalization test, and action distribution test. We also analyze the results and discuss some potential reasons based on the experiments we implemented.

5.1 PPO Training

The PPO training results is shown as Fig. 2, Fig. 3, and Fig. 4. The evaluation is conducted for the checkpoints reaching 10k, 100k, 200k, 300k, 400k, 500k, 600k, 700k, 800k, 900k, and 1000k training steps, shown in Fig. 5 and Fig. 6. As the training proceeded, we found different learning patterns between continuous and discrete action space settings. At the early stage of the training (before 200k training steps), the learning pattern is similar for both cases. However, starting from around 300k steps, the discrete case policy successfully completes the whole race track spending fewer actions, while the continuous case policy still suffers from a relatively low average reward with early termination of some episodes (the episode length is less than 1000, and gets lower). After that, generally, the discrete case policy performs better with a higher average reward and lower episode length, compared to the continuous case policy. At the end of the training reaching 1000k steps, the two policies can gain a similar level of average reward, while taking the same level of action steps to finish the task, shown in Fig. 5 and Fig. 6.

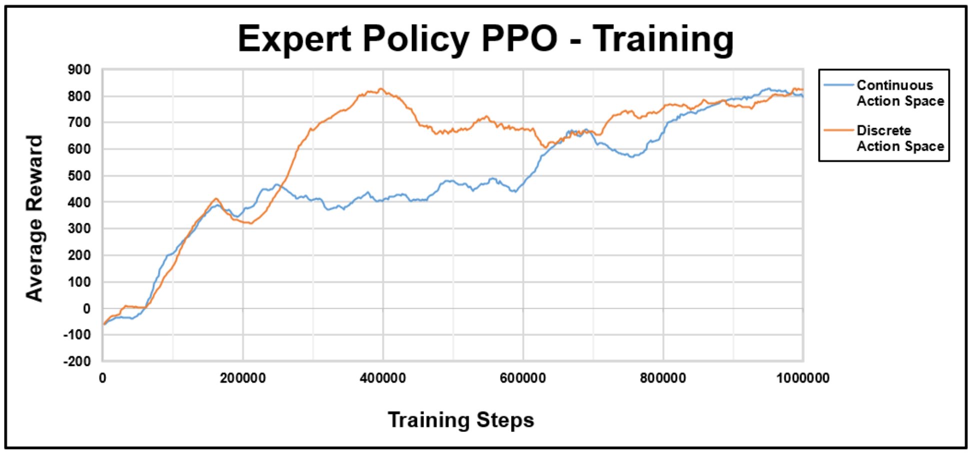

Figure 2: The PPO training results of the expert policy, show how the average reward changes during the process.

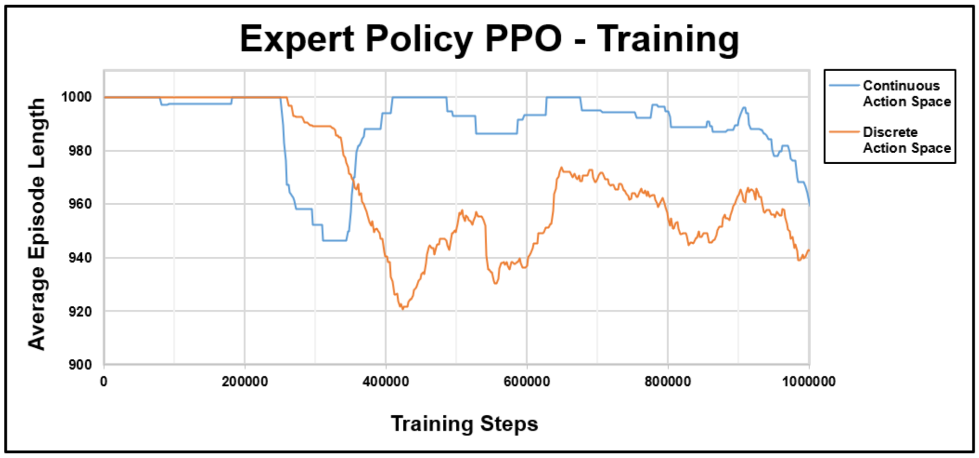

Figure 3: The PPO training results of the expert policy, show how the average episode length changes during the process.

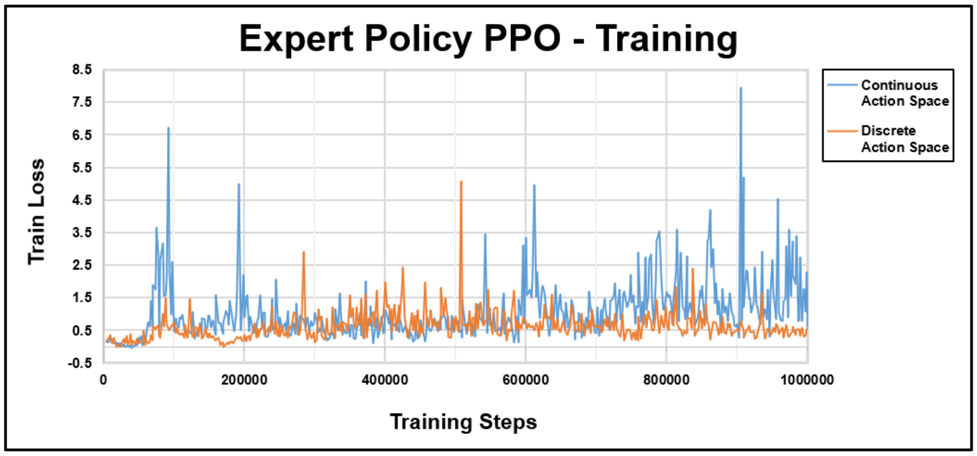

Figure 4: The PPO training results of the expert policy, show how the train loss changes during the process.

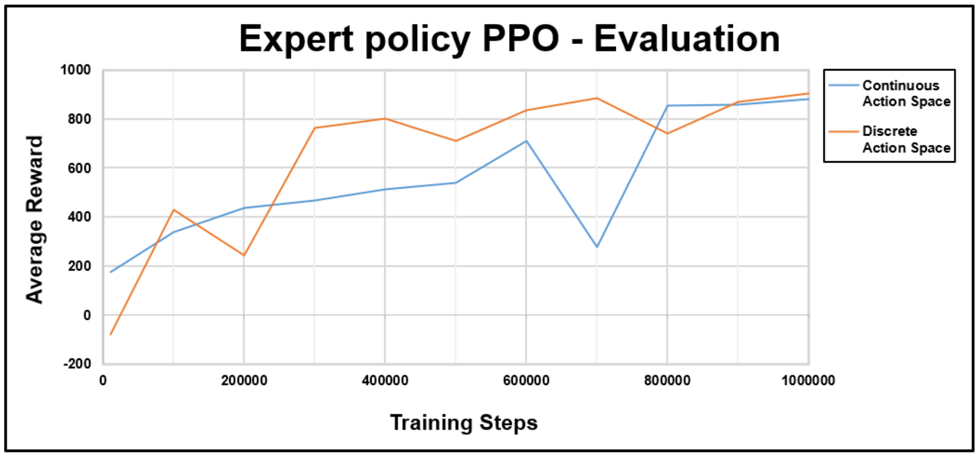

Figure 5: The PPO evaluation results, show how the average reward changes during the process. We run 20 episodes for each evaluation.

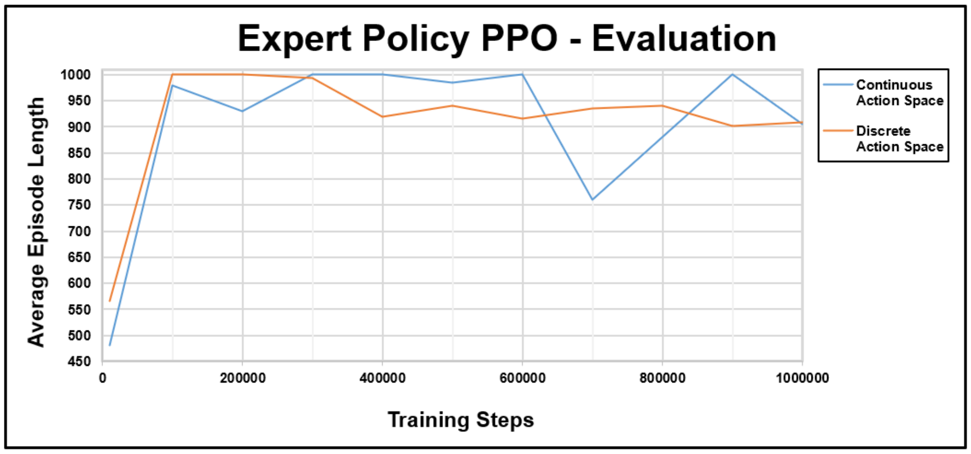

Figure 6: The PPO evaluation results, show how the average episode length changes during the process. We run 20 episodes for each evaluation.

We can infer from this pattern that the agent tends to learn aggressive driving skills when the action space is continuous, because it may run off the track each time it tries to perform drifting or speeding up too fast before the next curve. An interesting phenomenon for continuous cases is that the agent starts to mess up the task around 700k training steps with low rewards. The agent often even runs out of the playfield. However, the agent improves the skills for its dangerous driving style until 800k, where the agent still drives aggressively but succeeds more. The agent eventually performs as well as the discrete action space with a completely different driving pattern. The training loss for the discrete case remains more steady than the continuous case shown in Fig. 4, especially after 700k training steps.

Some techniques and special cases are shown in Fig. 7, including taking shortcuts, returning back to the track, passing through sharp curves, and encountering two curves. We also observed the driving patterns for continuous and discrete cases every 200k train steps until the training ends (700k training steps as a special case is observed as well).

At 200k train steps, the continuous case policy tends to drive on the outer (right-hand) side of the track at high speed. Its acceleration is too aggressive, so it often fails to adjust the position before a bend, causing the agent to run off the track and never return. For the discrete case, the policy drives extremely slowly with a strategy that is too conservative. The agent tends to drive on the inner side to

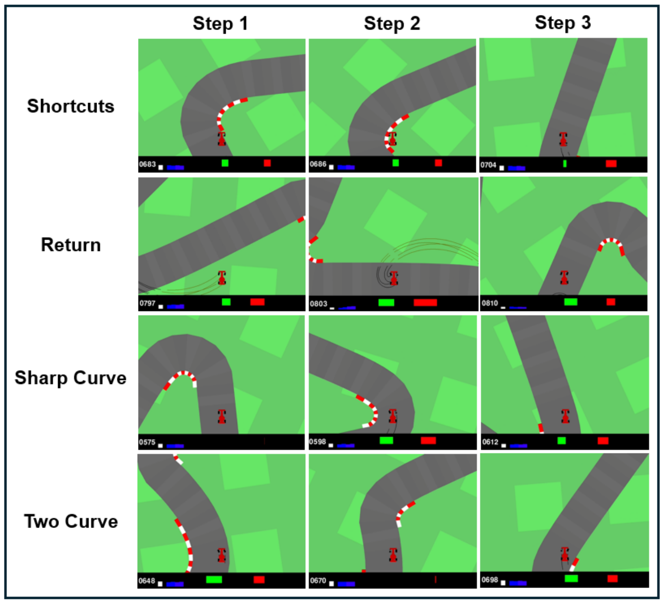

Figure 7: This figure shows some special driving skills and situations. We list 4 cases which are taking shortcuts, returning back to the track, passing a sharp curve, and driving through two curves.

pass through a bend and does not know how to adjust its position on straight tracks. Due to extremely slow speed, the agent succeeds in passing the bends but still earns a low reward.

At 400k and 600k train steps, the two different policies start improving themselves in different ways. The continuous case policy starts to learn how to slow down, but it still prefers to perform driving strategies. The agent starts to gain higher rewards and learn how to get back to the road after going off-track, but still cannot drive through all track tiles often. For discrete cases, the policy learns in the opposite way. The agent tries to speed up to get higher scores, but still drives in a conservative strategy using the brake frequently. Due to heavily relying on the brake, the agent still does not learn how to adjust its position before encountering a bend and may run off the track when there is a sharp one. At 700k, the continuous policy tries to drive even more aggressively, almost not using the brake and just relying on the friction to slow down. As a result, the agent often skids off the track and even hits the border of the playfield.

During 800k and 1000k, the two policies can complete all race track tiles with a high reward, while still holding their driving style. The cotinuous policy can perform accurate steering, while only using the friction to slow down without using the brake. It drives in an aggressive style and often uses techniques like drifting or frequent left or right steering. In contrast, the discrete case policy eventually drives in a conservative style with an appropriate frequency of acceleration and brake. This steady approach guarantees that off-track incidents are very rare and not too likely to happen.

We select the best checkpoint of 1000k training steps to be the expert policy for both continuous and discrete action space. The average rewards of the continuous and discrete cases are 879.23 and 904.22, which both are competitive records on the OpenAI official leaderboard.

5.2 BC Training

In BC training, trained expert policy is used to generate trajectories of demonstrations. We experimented with 1, 5, 10, 20, 30, 40, and 50 generated trajectories for supervised learning with CNN. The training results of BC are shown in Fig. 8, Fig. 9 for the smoothed train loss of continuous and discrete cases. The evaluation is shown in Fig. 10, where we also compared the performance between the MLP and CNN using a sufficient number of trajectories, which is 20, shown in Fig. 11.

In continuous cases, we found that the agent starts to have poor performance when providing more than 30 expert trajectories. One possible reason might be the higher level of noise when mapping the current policy with the expert policy, as we could not find this phenomenon in the discrete cases.

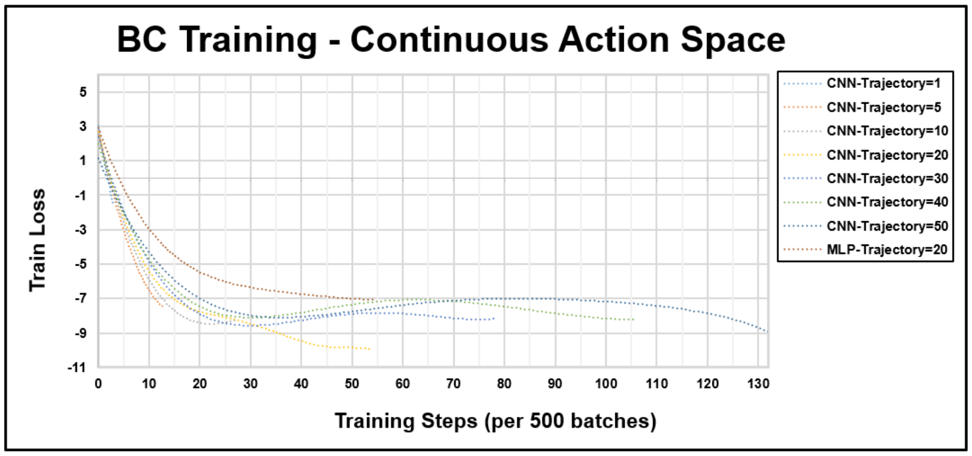

Figure 8: The training loss of the vanilla BC training for continuous action space. The curve is smoothed to represent a clear trend of the loss change.

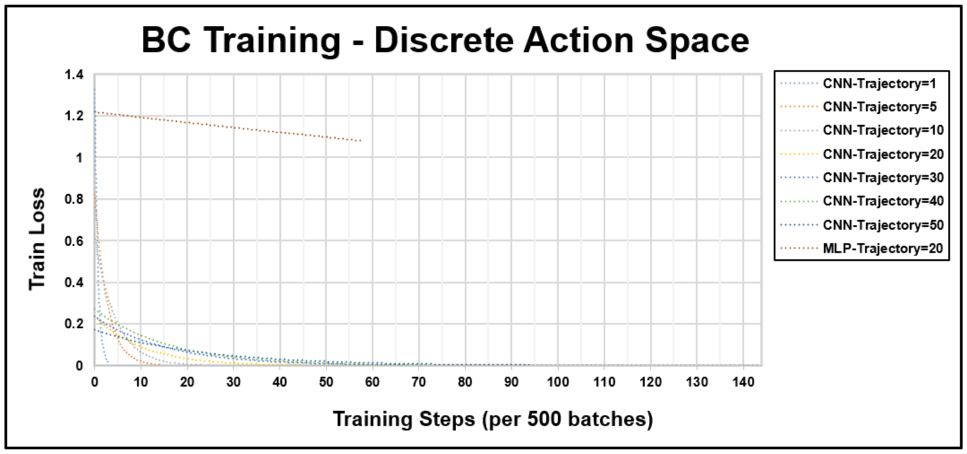

Figure 9: The training loss of the vanilla BC training for discrete action space. The curve is smoothed to represent a clear trend of the loss change.

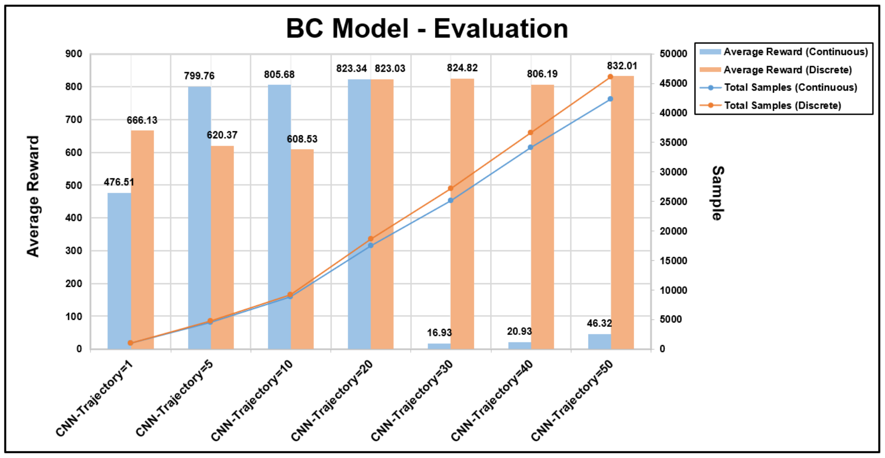

Figure 10: The evaluation results of the BC training, show the results of the average reward. The total samples are also shown as lines in this figure.

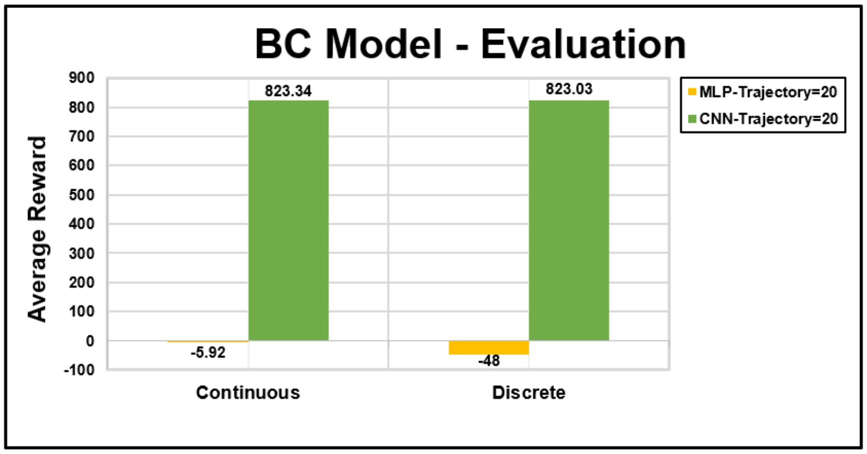

Figure 11: The comparison of the performance of using the MLP and CNN structure for BC training. The number of demonstration trajectories is fixed to 20 in this part.

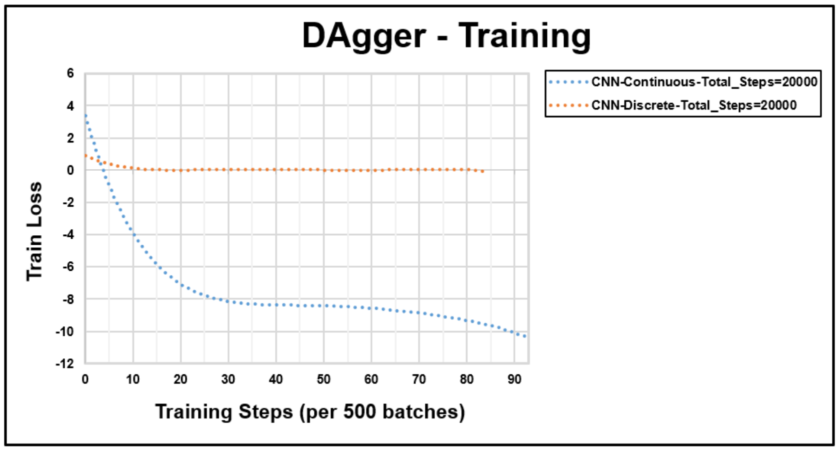

Figure 12: The training loss of the DAgger training for continuous and discrete action space. The curve is smoothed to represent a clear trend of the loss change. In this figure, each training step contains 500 batches.

However, as the number of trajectories increases, the marginal improvement of the reward becomes saturated. As a result, we selected 20 trajectories to compare the MLP and CNN structure. In Fig. 11, the results show that a CNN structure is far more suitable than an MLP for image inputs as observations.

5.3 DAgger Training

The training and evaluation results of DAgger training are shown in Fig. 12 and Fig. 13 respectively. We evaluated the performance of the checkpoint models for the total steps of 1k, 2k, 3k, 4k, 5k, 10k, and 20k. Similar to BC training, we also compared the MLP and CNN structure based on the same total training steps of 10k. The results show that the CNN structure is still a better choice for our environment settings. The performance of the DAgger policy reaches a high value when the total step is 10k, while only remaining at a similar level when continuing training to 20k.

5.4 Generalization Test

For the generalization test, we test the BC and DAgger policy again starting with a different seed value and running 20 episodes for different maps from what may have experienced during the training.

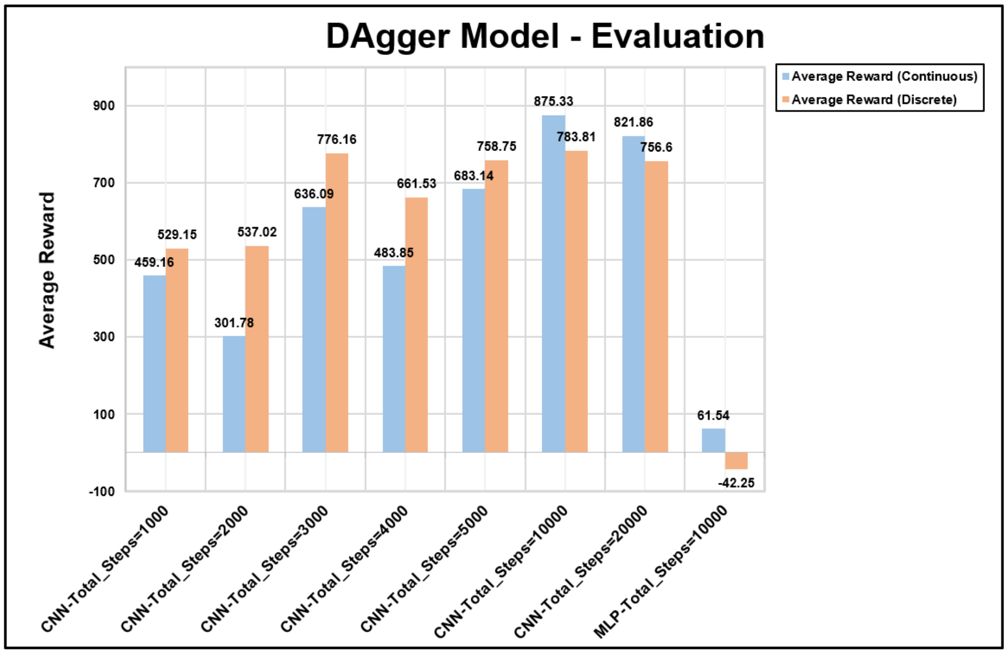

Figure 13: The evaluation results of the DAgger training, show the results of the average reward.

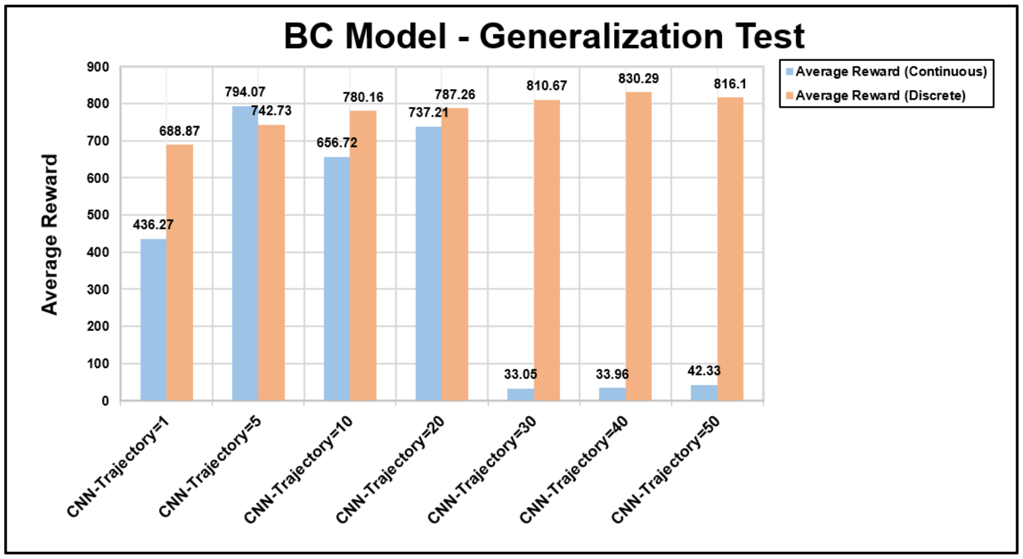

Figure 14: The performance of the generalization test for BC training. A different seed value from the training is used in this test.

In Fig. 14 and Fig. 15, both BC and DAgger policies generally remain similar trend and the same level of performance (or slightly better in some cases), which means that our expert policies also have the capability of making appropriate actions for unseen environments. This implies that our learning system and policies are robust enough and able to replicate its strategy to any random race track map provided by the simulator.

In Fig. 16, we selected the best PPO, BC, and DAgger policies based on the generalization test, comparing the performance gain and time cost. For the BC policy, we select 20 trajectories as the best BC policy training case. For continuous space, the performance achieves 85.5% of the best PPO policy (1000k training step). For discrete action space, the performance reaches 91.6%. If providing

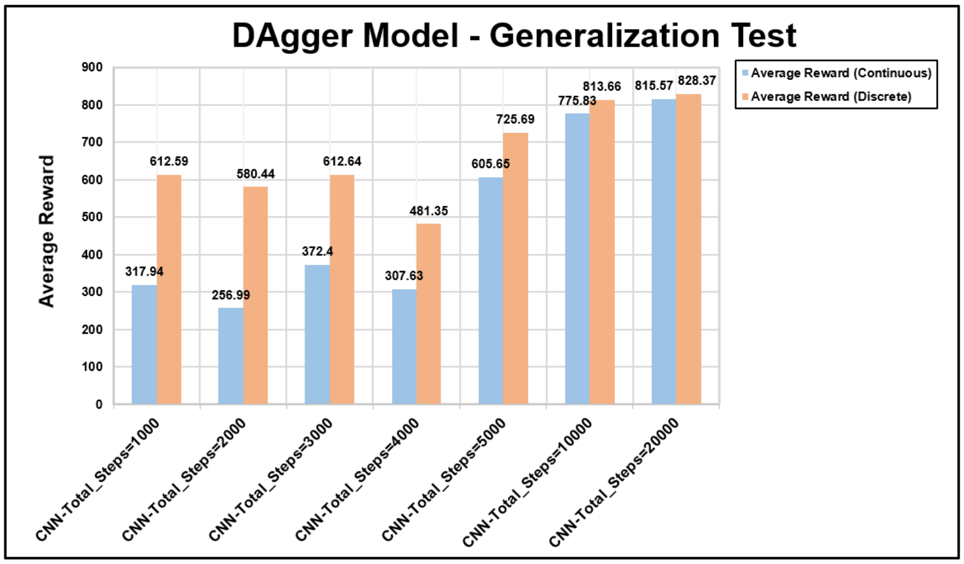

Figure 15: The performance of the generalization test for DAgger training. A different seed value from the training is used in this test.

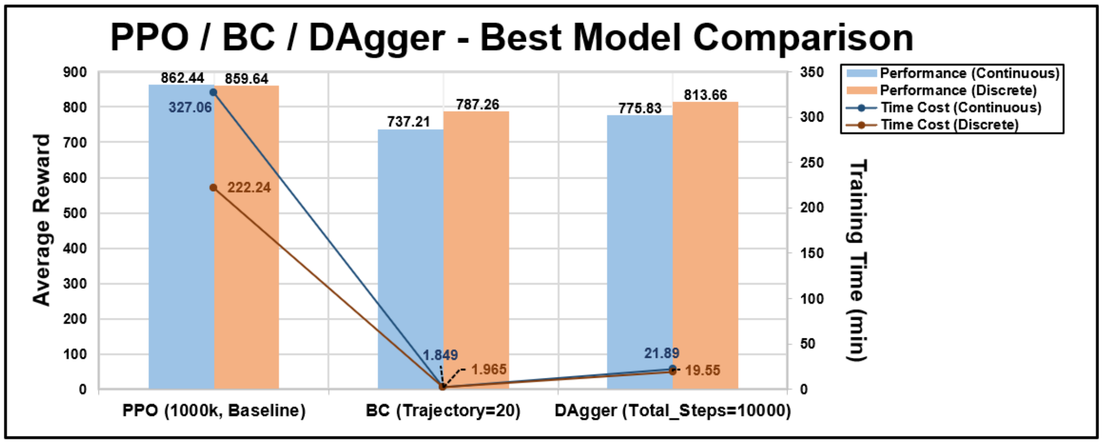

Figure 16: The comparison of performance and time cost between the best PPO, BC, and DAgger policies.

40 trajectories, the performance may improve to 96.6% (an increase of 5%) but has a trade-off of 196.6% of the time cost, which the cost is nearly doubled. This is the reason why we picked 20 trajectories instead of 40 for the best BC policy. For the DAgger policy, we picked the case of total steps to be 10k instead of 20k for the same reason. The performance reaches 90% and 94.7% when 10k total steps for continuous and discrete cases respectively. If keep training to 20k of total steps, the performance only increases to 94.6% (raises 4.6%) and 96.4% (raises only 1.7%) for continuous and discrete cases. The time cost increases by 308.1% and 372% for the two different cases. This is the reason why we eventually picked the 10k total steps to be the best DAgger policy.

Comparing the best PPO, BC, and DAgger policies, in continuous action space, the best BC and DAgger policy performs 85.5% and 90% of the best PPO policy under the same settings and same environments. The time cost can be reduced to 0.6% and 6.7% of the original PPO training time when only training the BC and DAgger policy. For discrete action space, the performances for the best BC and DAgger policies are 91.6% and 94.7%, while the time cost is only 0.9% and 8.8% respectively. As a result, if we only have access to the demonstrations for a certain task, vanilla BC would be a good attempt to make the policy work with an extremely low time cost, rather than train

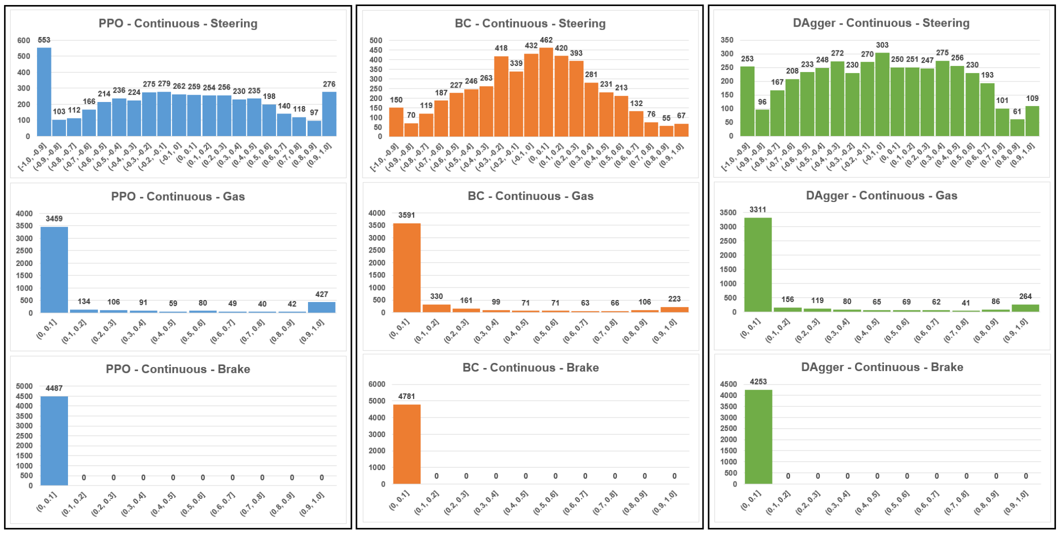

Figure 17: The action distributions of the best PPO (1000k train steps), BC (20 trajectories provided), and DAgger policies (20k total train steps) for continuous action space. Here we selected to use 20k total train steps for the DAgger policy since it performs slightly better than 10k and we are not considering the time cost in the action distribution test.

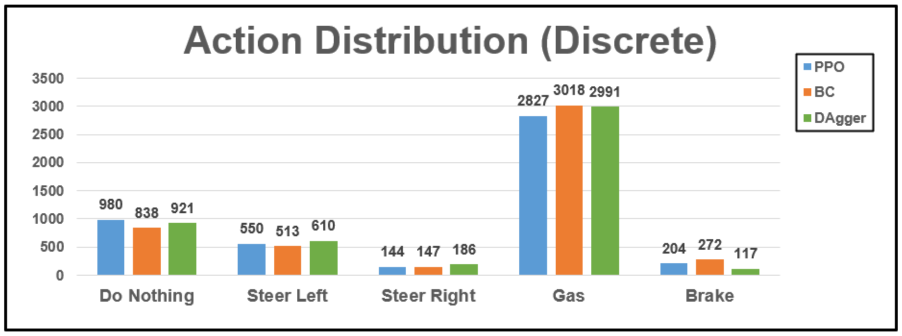

Figure 18: The action distributions of the best PPO (1000k train steps), BC (20 trajectories provided), and DAgger policies (20k total train steps) for discrete action space. Here we selected to use 20k total train steps for the DAgger policy since it performs slightly better than 10k and we are not considering the time cost in the action distribution test.

a policy using reinforcement learning. However, if we can get the expert policy itself, DAgger is a good choice to get a small improvement compared to vanilla BC, while we can still benefit from a relatively low time cost.

5.5 Action Distribution Test

The last test is the action distribution test, where we used the best PPO, BC, and DAgger policies from the previous generalization test (we decided to only change the DAgger policy using the 20k total train steps since it performs slightly better than 10k and we are not considering the time cost in this test). The agent runs 5 episodes while we record all its actions during the test. The results for continuous and discrete action space are shown in Fig. 17 and Fig. 18 respectively.

It is clear that the decision-making of action is easier to transfer from the PPO policy to the BC and DAgger policy in discrete action space. We can observe that the action distributions of the three policies are similar, compared to continuous action space. On the other hand, the action distributions of the three policies vary a lot in continuous action space, especially the BC policy. The DAgger policy is slightly closer to the original PPO policy, but still different to some extent. One major difference can be observed in the steering action, where the PPO policy tends to fully steer left or right, performing an aggressive driving style. In the BC policy, the agent prefers to slightly steer left or right more frequently while reducing full turns significantly. The DAgger policy acts more uniformly than the PPO and BC policy, having more normal steering patterns. Another pattern worthwhile to notice is that the agent in continuous space does not tend to use the brake, where the values are almost 0 for all three policies. However, in the discrete space, the agent is encouraged to use the brake to reduce the vehicle speed. This matches what we have seen in the observation of the PPO policy performance, where, in continuous settings, the agent tends to only use friction to slow down the car without triggering the brake.

6 Conclusion

We successfully built the whole pipeline to implement reinforcement and imitation learning for the car racing environment. All policies have a performance of over 85% of the baseline PPO policy while having less than 10% of the baseline time cost. The best policy with the highest performance is trained through DAgger in discrete action space, reaching 94.7% of the PPO policy performance. We found how efficient the BC and DAgger algorithm can be, compared to the time cost of the PPO algorithm. The BC policies in continuous and discrete settings can have less than 1% of the baseline time cost, while both still preserving over 85% of the PPO policy performance. We trained two series of policies performing different driving strategies with continuous and discrete action space, the former is aggressive and the latter is conservative. Through the analysis of the action distribution test, we discovered how strategies transfer through imitation learning, while the pattern is very different between continuous and discrete cases. To sum up, this project aims to study the pros and cons of reinforcement learning and imitation learning through implementation. Although using simple algorithms like vanilla BC and DAgger, the results give us a clear picture of when we should attempt to use which algorithm for our task.

This project was worked within a short term, while we tried to include the complete pipeline and a series of tests for evaluation from different aspects. However, if we have 3 additional months or more time, we can continue to explore deeper topics. For example, we can work on different environments with a similar observation format, such as other Atari games, exploring the possibilities of transfer learning with the help of imitation learning. Another direction is to work on more learning algorithms using demonstrations or expert policies, such as Generative Adversarial Imitation Learning (GAIL) or Inverse reinforcement learning (IRL). We may learn more specific details of the advantages and drawbacks through actual implementation and analysis.

References

[1] Arthur L Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of research and development*, 3(3):210–229, 1959.

[2] Marvin Minsky. Steps toward artificial intelligence. *Proceedings of the IRE*, 49(1):8–30, 1961. doi: 10.1109/JRPROC.1961.287775.

[3] Gavin A Rummery and Mahesan Niranjan. *On-line Q-learning using connectionist systems*, volume 37. University of Cambridge, Department of Engineering Cambridge, UK, 1994.

[4] Peter Dayan and CJCH Watkins. Q-learning. *Machine learning*, 8(3):279–292, 1992.

[5] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8:229–256, 1992.

[6] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[7] Oleksii Zhelo, Jingwei Zhang, Lei Tai, Ming Liu, and Wolfram Burgard. Curiosity-driven exploration for mapless navigation with deep reinforcement learning. *arXiv preprint arXiv:1804.00456*, 2018.

[8] Victor Talpaert, Ibrahim Sobh, B Ravi Kiran, Patrick Mannion, Senthil Yogamani, Ahmad El-Sallab, and Patrick Perez. Exploring applications of deep reinforcement learning for real-world autonomous driving systems. *arXiv preprint arXiv:1901.01536*, 2019.

[9] Oriol Vinyals, Igor Babuschkin, Wojciech M Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H Choi, Richard Powell, Timo Ewalds, Petko Georgiev, et al. Grandmaster level in starcraft ii using multi-agent reinforcement learning. *nature*, 575(7782):350–354, 2019.

[10] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[11] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *International conference on machine learning*, pages 1889–1897. PMLR, 2015.

[12] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[13] Sergey Levine, Aviral Kumar, George Tucker, and Justin Fu. Offline reinforcement learning: Tutorial, review, and perspectives on open problems. *arXiv preprint arXiv:2005.01643*, 2020.

[14] Dean A Pomerleau. Efficient training of artificial neural networks for autonomous navigation. *Neural computation*, 3(1):88–97, 1991.

[15] Faraz Torabi, Garrett Warnell, and Peter Stone. Behavioral cloning from observation. *arXiv preprint arXiv:1805.01954*, 2018.

[16] Stéphane Ross, Geoffrey Gordon, and Drew Bagnell. A reduction of imitation learning and structured prediction to no-regret online learning. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 627–635. JMLR Workshop and Conference Proceedings, 2011.

[17] Pieter Abbeel and Andrew Y Ng. Apprenticeship learning via inverse reinforcement learning. In *Proceedings of the twenty-first international conference on Machine learning*, page 1, 2004.

[18] Jonathan Ho and Stefano Ermon. Generative adversarial imitation learning. *Advances in neural information processing systems*, 29, 2016.