Reinforcement Learning for LunarLander-v3: A Comprehensive Approach Using Imitation Learning and Self-Play

Siddharth Vikram
IMT2022534

Vaibhav Bajoriya
IMT2022574

Tanish Pathania
IMT2022049

May 10, 2025

Introduction

Motivation

In complex decision-making environments such as single-player games, developing autonomous agents that learn efficiently from limited supervision remains a key challenge. Unlike multiplayer settings, single-player scenarios lack external adversarial dynamics, requiring more sophisticated methods to drive exploration and skill acquisition. This project is motivated by the goal of leveraging imitation learning—specifically behavioral cloning and DAgger—to bootstrap an agent's performance from expert demonstrations in the LunarLander-v3 environment. We further enhance this with a novel form of self-play, where successive generations of the agent compete against earlier versions of themselves. This approach allows for continual refinement through internal competition and multi-objective optimization, resulting in a more robust and generalizable policy without reliance on hand-crafted rewards or dense supervision.
This hybrid approach offers the best of both paradigms: the sample efficiency and safety of supervised learning combined with the open-ended optimization of competitive self-play. By using expert demonstrations to establish fundamental competence and self-play to refine and expand capabilities, we can develop robust agents that not only successfully land the lunar module but potentially surpass the performance of the original expert demonstrators.

General Environments for Imitation Learning and Self-Play

The imitation learning and self-play methodology we have implemented can be effectively applied to a broad range of environments beyond *LunarLander*, provided they exhibit certain key characteristics.

Suitable Environment Characteristics

- Complex decision-making tasks where random exploration is highly inefficient.
- Sparse or delayed reward structures that make traditional reinforcement learning challenging.
- Availability of expert demonstrations, either from human experts or pretrained agents.
- Tasks with clear success or failure outcomes, enabling comparative performance analysis between agents.
- Presence of a competitive element or the potential for meaningful policy comparisons.

Use Cases and Benefits

This methodology is particularly effective when:

- Initial random exploration may lead to unsafe or inefficient learning trajectories.
- The task exhibits a clear progression in skill levels.
- Comparing different policies provides a meaningful learning signal.

By leveraging demonstration data to bootstrap learning, followed by iterative self-play for policy refinement, this approach enables rapid and robust policy development beyond what initial demonstrations alone could provide.

Real-World Applications

While LunarLander is a simulation environment, the techniques developed in this project have direct applications to numerous real-world problems:

- **Spacecraft Landing Systems**: The most direct application is autonomous landing systems for lunar and planetary missions. NASA, SpaceX, and other space agencies are actively developing autonomous landing capabilities for the Moon, Mars, and beyond. Our approach could reduce development time and increase landing reliability.
- **VTOL Aircraft Control**: Vertical Take-Off and Landing (VTOL) aircraft face similar control challenges involving precise thrust management and stabilization. Companies developing air taxis and urban air mobility solutions could benefit from our hybrid learning approach.
- **Robotic Manipulation**: Industrial robotic arms performing precision assembly tasks encounter similar challenges in controlled motion with multiple degrees of freedom. Our methodology's sample efficiency is particularly valuable when collecting real-world robot data is expensive.
- **Drone Delivery Systems**: Autonomous package delivery drones must navigate complex environments and perform precise landings in varied conditions. The robustness developed through our self-play approach would be valuable for handling unexpected scenarios.

In each of these applications, the combination of imitation learning (to safely bootstrap from expert demonstrations) and self-play (to exceed expert performance) addresses the critical challenges of exploration safety, sample efficiency, and continuous improvement.

MDP Formulation

We formally model the LunarLander problem as a Markov Decision Process (MDP), defined by the tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$:

- **State Space** $\mathcal{S}$: An 8-dimensional continuous vector $s \in \mathbb{R}^8$ where:
    - s_1, s_2: Coordinates of the lander relative to the landing pad (normalized to approximately $[-1, 1]$)
    - s_3, s_4: Linear velocities in x and y directions (normalized to approximately $[-1, 1]$)
    - s_5: Angle of the lander (normalized to approximately $[-\pi, \pi]$)
    - s_6: Angular velocity (normalized)
    - s_7, s_8: Boolean flags indicating contact between each leg and the ground
- **Action Space** $\mathcal{A}$: A discrete set of 4 actions $\mathcal{A} = \{0, 1, 2, 3\}$ where:
    - $a = 0$: Do nothing
    - $a = 1$: Fire left orientation engine
    - $a = 2$: Fire main engine
    - $a = 3$: Fire right orientation engine
- **Transition Dynamics** $\mathcal{P}(s'|s, a)$: The probabilistic state transition function determined by Box2D physics simulation. Given the current state s and action a, the environment transitions to a new state s' according to:

  $$s' = f(s, a) + \epsilon \tag{1}$$

  where f represents the deterministic physics equations and ϵ accounts for small amounts of environmental noise.
- **Reward Function** $\mathcal{R}(s, a, s')$: The reward function combines several components:

  $$\mathcal{R}(s, a, s') = R_{\text{shaping}}(s, s') + R_{\text{action}}(a) + R_{\text{terminal}}(s') \tag{2}$$

  where:

  $$R_{\text{shaping}}(s, s') = \text{reward for getting closer to landing pad} \tag{3}$$

  $$R_{\text{action}}(a) = -0.3 \cdot \mathbb{I}[a = 2] \quad \text{(fuel cost for main engine)} \tag{4}$$

  $$R_{\text{terminal}}(s') = \begin{cases} +100 & \text{if successful landing} \\ -100 & \text{if crash} \end{cases} \tag{5}$$
- **Discount Factor** γ: Set to 0.99, balancing immediate and future rewards while ensuring the cumulative reward remains bounded.

The goal of our agent is to find an optimal policy $\pi^* : \mathcal{S} \rightarrow \mathcal{A}$ that maximizes the expected discounted cumulative reward:

$$\pi^* = \arg\max_{\pi} \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t \mathcal{R}(s_t, \pi(s_t), s_{t+1})\right] \tag{6}$$

Methodology

Our approach combines expert demonstration, imitation learning, and reinforcement learning through the following methods:

Architecture

Our implementation of the neural network architecture follows:

```
class DQN(nn.Module):
    def __init__(self, n_observations, n_actions):
        super().__init__()
        self.layer1 = nn.Linear(n_observations, 128)
        self.layer2 = nn.Linear(128, 128)
        self.layer3 = nn.Linear(128, n_actions)

    def forward(self, x):
        x = F.relu(self.layer1(x))
        x = F.relu(self.layer2(x))
        return self.layer3(x)
```

Deep Q-Networks for Expert Policy

We first trained an expert policy using DQN with the following key components:

- Experience replay buffer to store transitions
- Target network for stable Q-learning
- Epsilon-greedy exploration strategy with annealing
- Double DQN to reduce overestimation bias

Behavioral Cloning

Using collected expert trajectories, we trained a student model to mimic the expert's behavior by minimizing the cross-entropy loss between the student's action distributions and the expert's actions.

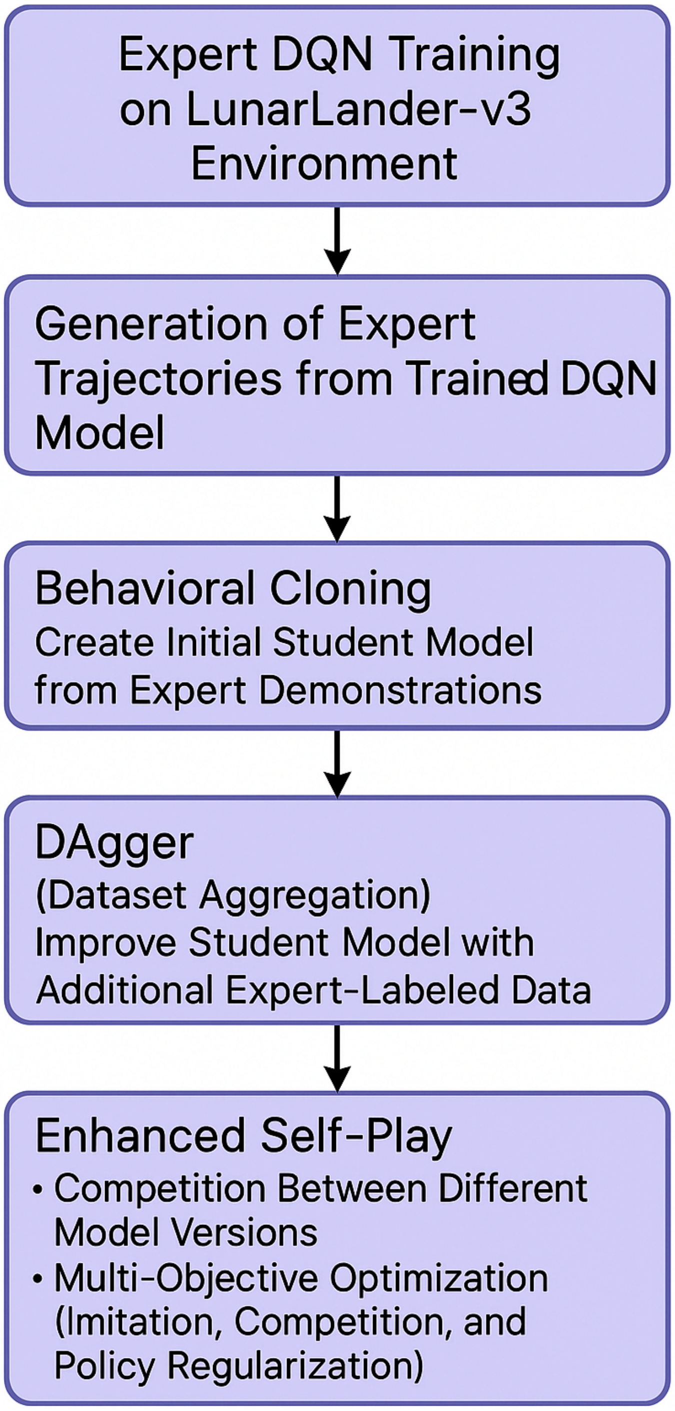

Expert DQN Training on LunarLander-v3 Environment
Generation of Expert Trajectories from Trained DQN Model
Behavioral Cloning
Create Initial Student Model from Expert Demonstrations
DAgger
(Dataset Aggregation)
Improve Student Model with Additional Expert-Labeled Data
Enhanced Self-Play
• Competition Between Different Model Versions
• Multi-Objective Optimization (Imitation, Competition, and Policy Regularization)

Figure 1: Pipeline for ImitationLearning+SelfPlay

Dataset Aggregation (DAgger)

We implemented DAgger to address the covariate shift problem in behavioral cloning:

1. Initial training via behavioral cloning
2. Execution of student policy to collect states
3. Querying expert for action labels on visited states
4. Aggregation of new data with existing dataset
5. Retraining student on the augmented dataset

Self-Play with Previous Version

To further improve performance beyond expert demonstrations, we developed a self-play mechanism:

- Competition against previous model versions
- Strategic opponent selection based on learning value
- Enhanced reward system with competitive and safety components
- Multi-objective loss function combining:
  - Standard TD loss for RL
  - Cross-entropy loss for imitation
  - KL divergence for policy regularization
  - Safety penalties for catastrophic actions
  - Entropy bonuses for exploration

Usage instructions

- run the notebook IL+SelfPlay-WithPreviousVersion.ipynb

Self-Play with MCTS

This section outlines a single-agent reinforcement learning architecture combining *Monte Carlo Tree Search* (MCTS) with a policy network trained via *self-play imitation*. The target environment is `LunarLander-v3`, a discrete control task from Gymnasium. Unlike adversarial games, here the agent learns to improve through iterative self-play against a fixed environment by planning with MCTS and learning via behavior cloning.

System Architecture

The full pipeline is composed of the following components:

- **Policy Network (DQN)** – A neural network trained to mimic the action distribution produced by MCTS.
- **MCTS Engine** – A tree-based search algorithm using policy-guided rollouts.
- **Replay Buffer** – Stores self-play trajectories for supervised learning.
- **Self-Play Loop** – Iteratively generates experience, trains the policy, and evaluates progress.

Policy Network The policy is parameterized by a deep neural network with three fully connected layers. The network maps observations to action logits:

```
class DQN(nn.Module):
    def __init__(self, n_observations, n_actions):
        super().__init__()
        self.layer1 = nn.Linear(n_observations, 128)
        self.layer2 = nn.Linear(128, 128)
        self.layer3 = nn.Linear(128, n_actions)
```

This network outputs unnormalized logits for each discrete action. The MCTS uses these logits to bias its rollouts and expansions.

MCTS with Policy-Guided Rollouts Each decision step invokes an MCTS search from the current state. The tree is built by simulating actions and selecting children using the PUCT formula. Node expansion uses softmax probabilities from the policy network:

```
state_tensor = torch.FloatTensor(state).unsqueeze(0).to(device)
action_probs = torch.softmax(self.policy_net(state_tensor), dim=1).cpu().numpy().flatt
root.expand(action_probs, self.env)
```

During rollout:

```
action = np.random.choice(len(action_probs), p=action_probs)
next_state, reward, terminated, truncated, _ = env_copy.step(action)
```

Self-Play Episode Generation Self-play is implemented by simulating full episodes using the current MCTS policy. The actions chosen are logged and added to a replay buffer:

```
state, _ = self.env.reset()
while not done:
    action = self.mcts.get_action(state)
    self.replay_buffer.append((state.copy(), action))
    state, _, terminated, truncated, _ = self.env.step(action)
    done = terminated or truncated
```

Supervised Policy Update The policy network is updated using cross-entropy loss against the MCTS-selected actions:

```
states, actions = zip(*batch)
logits = self.policy_net(torch.FloatTensor(states).to(device))
loss = F.cross_entropy(logits, torch.LongTensor(actions).to(device))
```

Evaluation and Best Model Tracking After each generation, the agent is evaluated by acting greedily from its policy. The best-performing model is checkpointed:

```
if eval_reward > best_reward:
    torch.save(self.policy_net.state_dict(), "best_policy_net.pth")
```

Visualization Training metrics such as loss and reward curves are plotted using `matplotlib`, and action distributions are plotted with `seaborn`. Video recordings of gameplay before and after training are created using Gymnasium's `RecordVideo` wrapper.

```
env = gym.make(env_name, render_mode="rgb_array")
env = gym.wrappers.RecordVideo(env, video_folder="videos", name_prefix="final_policy")
```

Key Advantages

- **Improved Exploration:** MCTS provides lookahead capabilities to guide policy improvement even when rewards are sparse.
- **Stability:** Learning is supervised, avoiding instability from TD-based value learning.
- **Interpretability:** MCTS allows inspection of visit counts, action distributions, and decision paths.

Limitations

- **Computational Overhead:** Each MCTS step involves deep copies of the environment and many forward passes through the network.
- **Single-Step Planning:** The MCTS tree is reset at each timestep, which is suboptimal compared to persistent trees.

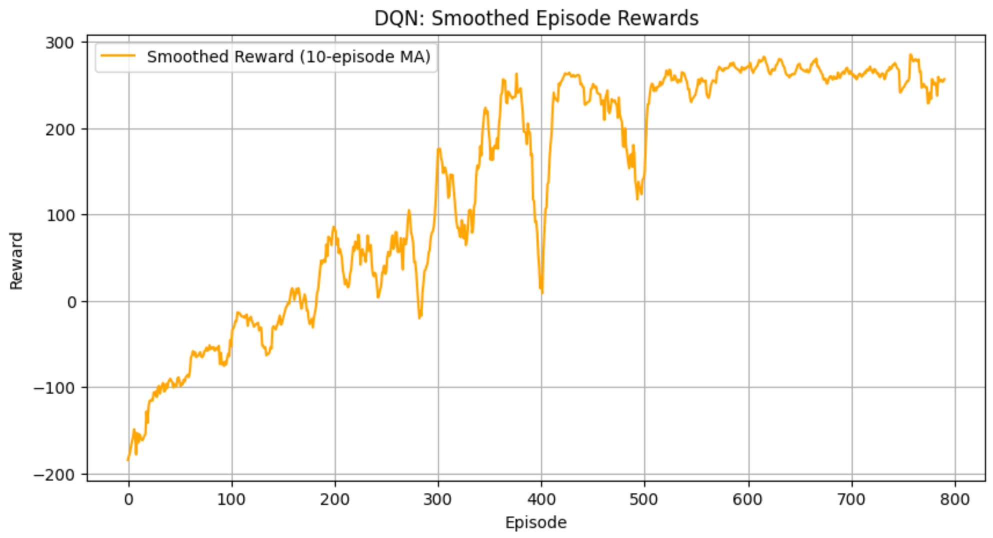
Usage instructions

- run the notebook **RL_mcts_self_play** with the imports stated below
- MCTS and single player self play pipeline python notebook requires pretrained dagger models. You can run the notebook with just one model, but 5 iterations of dagger models were used for experimentation. So for a complete run without code change, import the all the models of the form **dagger_model_iter.pth**

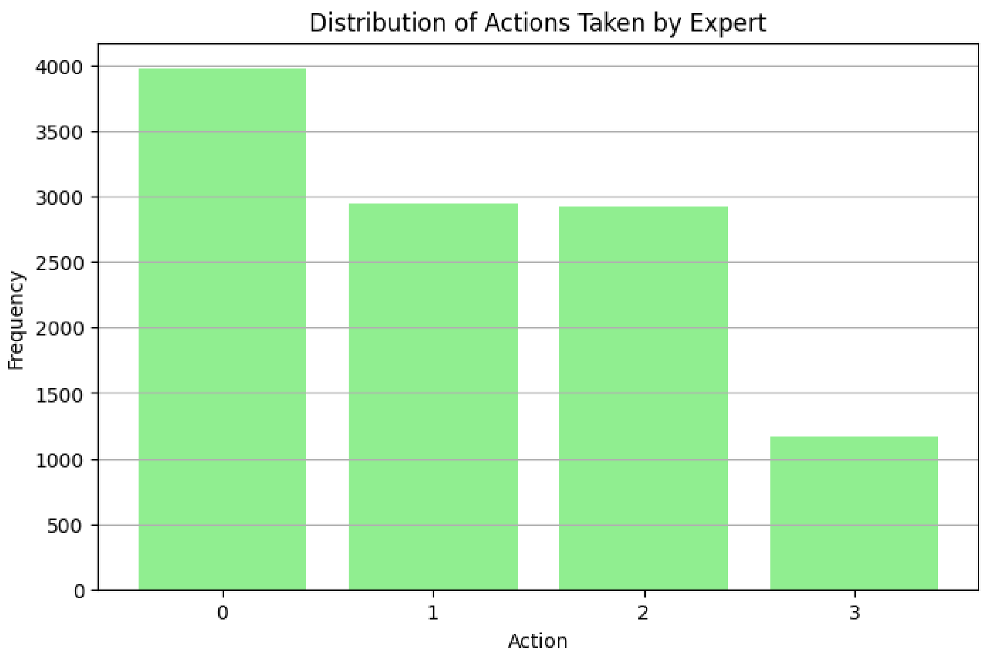
Experiments and Results

Evaluation Setup

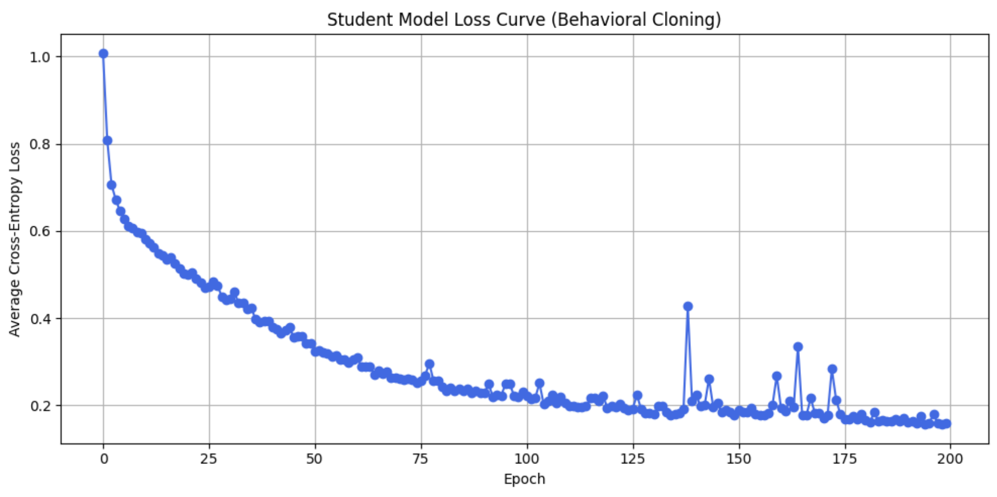
All four models—Expert DQN, Behavioral Cloning, DAgger, and Self-Play—were evaluated on 20 independent episodes of `LunarLander-v3` using `rgb_array` rendering. Each policy was loaded from its respective checkpoint and executed in greedy mode. Total episodic rewards were recorded to assess average performance and variability.



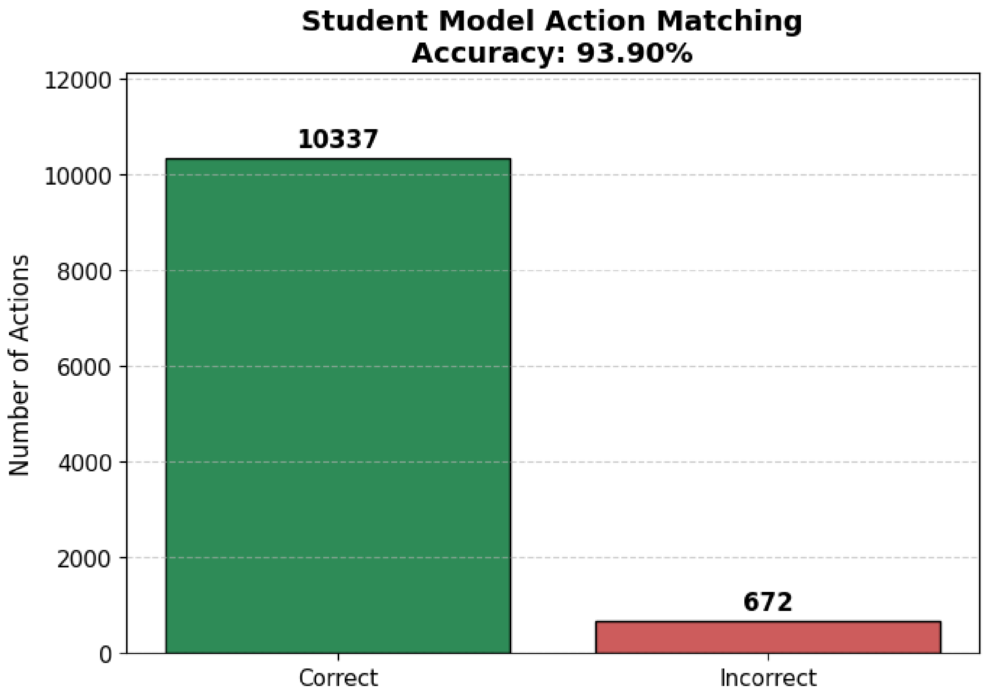
DQN Smoothed Episode Rewards. The plot illustrates the learning progression of the agent trained using the Deep Q-Network (DQN) algorithm. Each point represents the average episode reward over a window of recent episodes, smoothed to reduce variance and highlight the overall trend. The upward trajectory in the smoothed rewards indicates that the agent is successfully learning to maximize returns over time. Periodic fluctuations can be attributed to exploration-exploitation trade-offs and the stochastic nature of the environment.



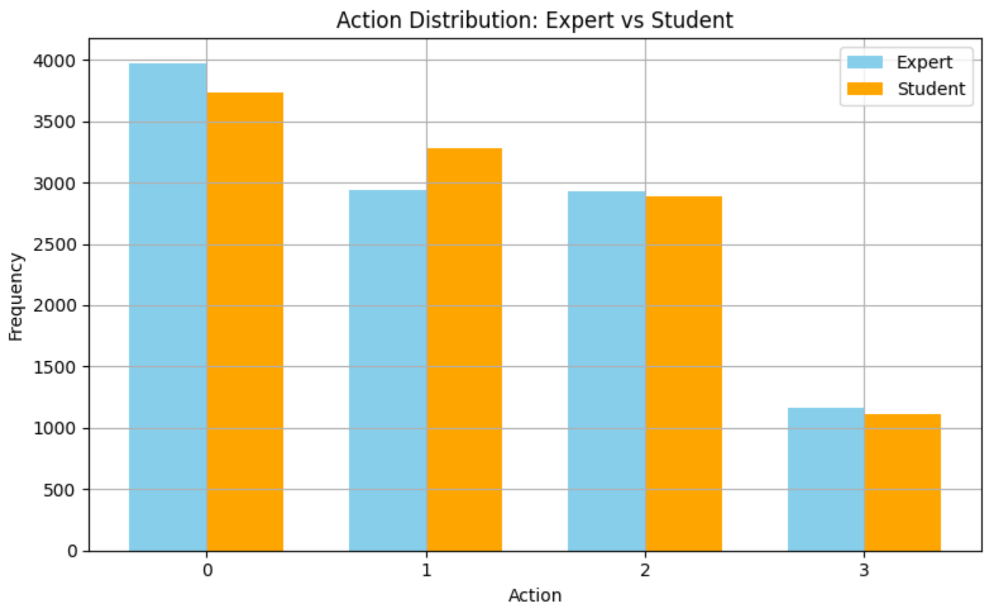
Distribution of Actions Taken by Expert. This histogram displays the frequency of each discrete action selected by the expert policy during demonstrations. Analyzing this distribution provides insight into the expert's behavioral tendencies and strategic preferences when navigating the environment. A skewed distribution may indicate dominant strategies or preferred maneuvers, which can guide imitation learning and policy initialization for reinforcement learning agents. Understanding this behavior is crucial for benchmarking and aligning learner performance with expert behavior.



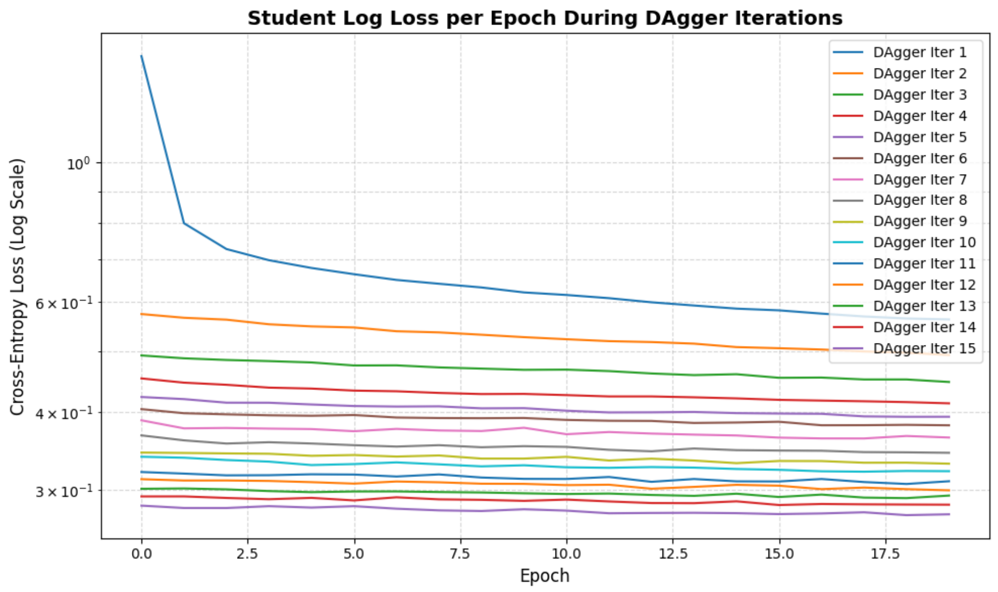
Student Log Loss per Epoch During DAgger Iterations. This plot tracks the cross-entropy loss (log loss) of the student model during each epoch across multiple DAgger (Dataset Aggregation) iterations. The decreasing trend in the loss indicates that the student model is increasingly aligning its action predictions with the expert's decisions. Occasional spikes may correspond to the inclusion of more diverse or challenging states in the aggregated dataset, prompting further learning. This metric serves as a key indicator of behavioral cloning efficiency and convergence.



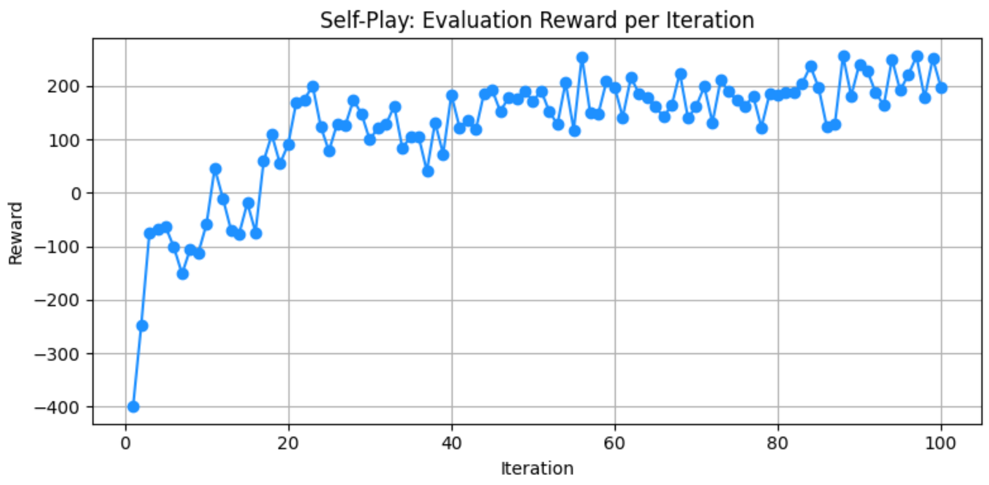
Student Model Action Matching Accuracy. This figure shows the percentage of actions predicted by the student model that match those of the expert across DAgger iterations. A rising trend in accuracy demonstrates improved imitation of expert behavior, reflecting successful knowledge transfer. As the student is exposed to more diverse states through interactive rollouts, this metric helps evaluate how effectively the student internalizes expert strategies. Near-perfect accuracy suggests strong behavioral cloning, while plateaus may indicate areas needing further refinement.



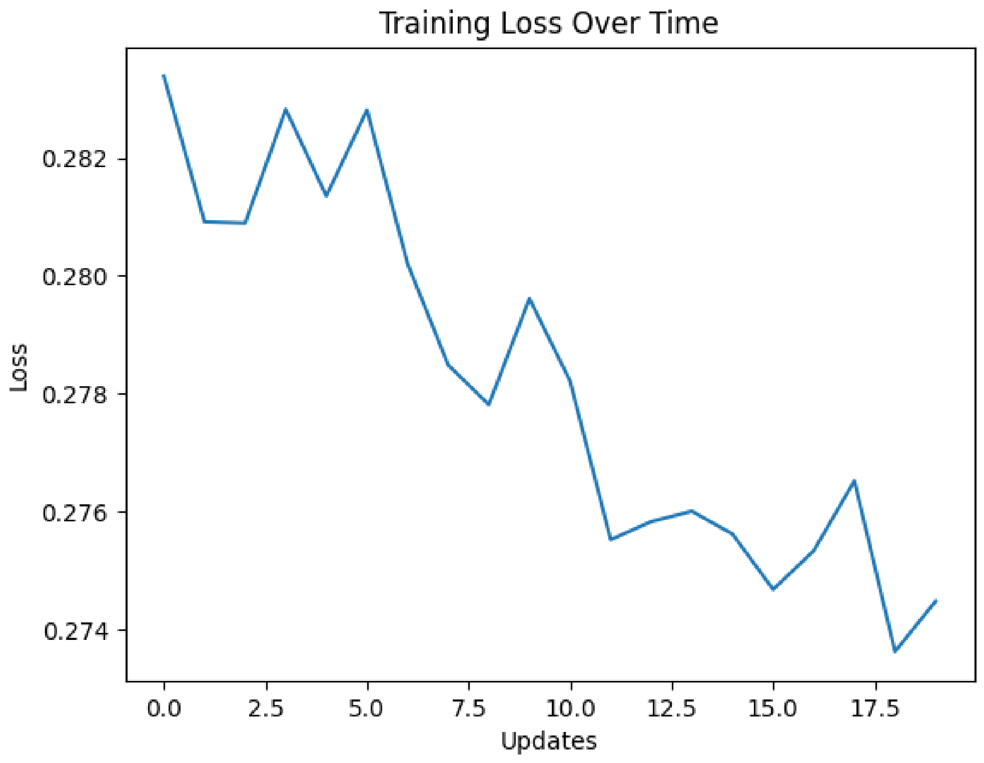
Action Distribution, Expert vs Student. This comparative bar chart illustrates the frequency of each discrete action taken by the expert and the student policy. Aligning the student's action distribution with that of the expert indicates effective imitation and successful behavioral cloning. Discrepancies between the two distributions may reveal biases in the student's learning or indicate areas where the student diverges from expert behavior. Such analysis is crucial for diagnosing model behavior beyond standard accuracy metrics.



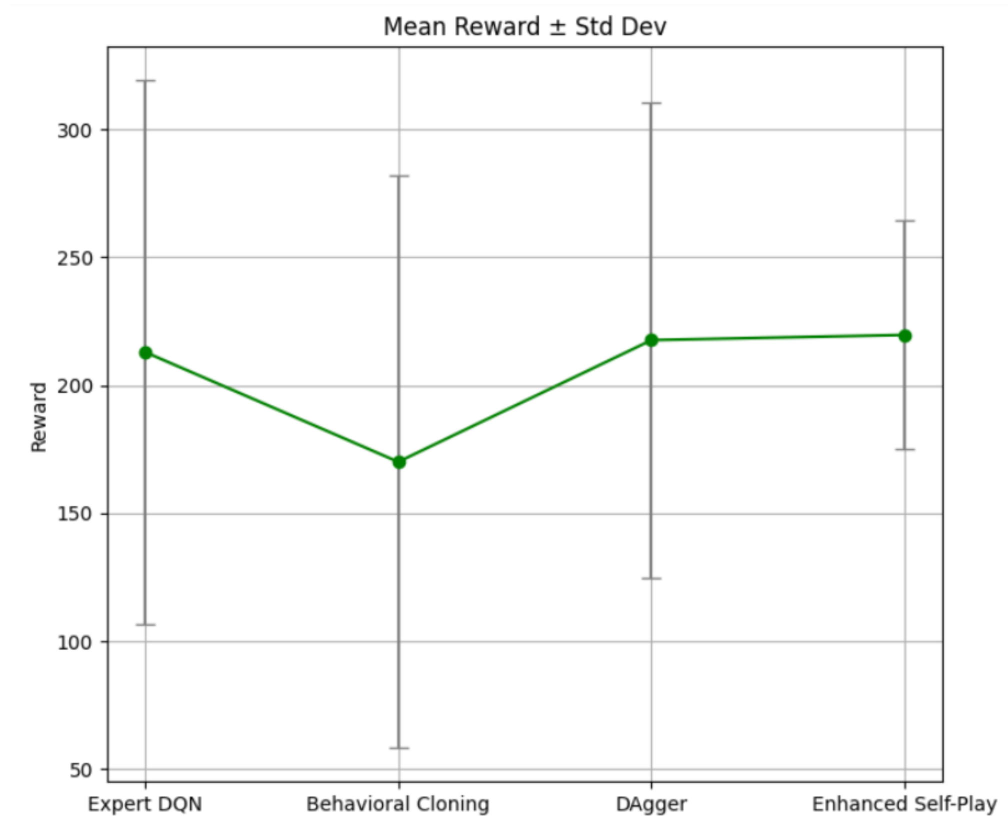
Student Log Loss per Epoch During DAgger Iterations. This plot offers an alternative view of the cross-entropy loss incurred by the student model while learning from the expert via DAgger. Each epoch reflects training progress using the aggregated dataset at that point. Consistent reduction in log loss indicates convergence toward expert-like action predictions. This visualization complements performance metrics by providing insight into the student's internal learning dynamics over time.



Self Play Evaluation Reward. This figure presents the evaluation rewards obtained by the agent during self-play across training iterations. The trend reflects the agent's performance as it competes against past versions of itself, enabling continuous policy refinement. An upward trajectory in rewards indicates progressive learning and improved policy robustness. Plateaus or dips may highlight challenges in surpassing earlier policy baselines or encountering suboptimal training dynamics. This metric is central to assessing policy improvement in a self-play setting.



Self-Play Training Loss Over Time. This plot shows the evolution of the training loss during self-play updates. The loss typically reflects the divergence between predicted action values and target values, as computed through Temporal Difference (TD) learning or other value-based methods. A declining loss suggests stable learning and better policy convergence. Fluctuations can occur due to changing opponents, exploration noise, or non-stationary dynamics introduced by self-play. Monitoring this loss is essential to ensure stable and effective policy learning.



Final comparison of all models. This figure compares the performance of various models trained using different learning paradigms, such as pure reinforcement learning (DQN), imitation learning (DAgger), and self-play. The comparison is based on a common evaluation metric—typically average episode reward or task success rate—under identical environmental conditions. This holistic visualization highlights the relative strengths and weaknesses of each approach, demonstrating which models achieve better generalization, robustness, and overall policy quality. Such comparisons are critical for evaluating the trade-offs between sample efficiency, stability, and performance across learning strategies.

Model Comparison

Model performance is summarized by mean±standard deviation of total rewards:

Model	**Mean Reward**	**Std. Dev.**
Expert DQN	246.09	83.56
Behavioral Cloning	246.72	69.29
DAgger	260.92	34.09
Self-Play	157.73	71.91

DAgger achieves the highest mean reward and lowest variance, while Self-Play underperforms relative to the expert-based methods in this evaluation.

Training Results

Loss Trajectories

- **Behavioral Cloning:** Cross-entropy loss declined monotonically over 100 epochs, indicating stable supervised learning.
- **DAgger:** Loss per iteration decreased steadily, confirming effective incorporation of expert feedback.
- **Self-Play:** Temporal-difference loss stabilized after early fluctuations, demonstrating convergence under on-policy updates.

Reward Curves

- **Expert DQN:** 50-episode moving average exceeded 200 consistently.
- **DAgger:** Rapid convergence to high rewards with low variability.
- **Self-Play:** Slower improvement and lower peak returns (mean 157.73) than expert-initialized methods.

MCTS Results

Training Analysis and Visualization

To monitor the training progress and policy behavior, we visualize two key diagnostics:

- **Training Loss and Evaluation Reward:** Tracks policy improvement and generalization over generations.
- **Action Distribution:** Summarizes which actions were favored by MCTS over the course of training.

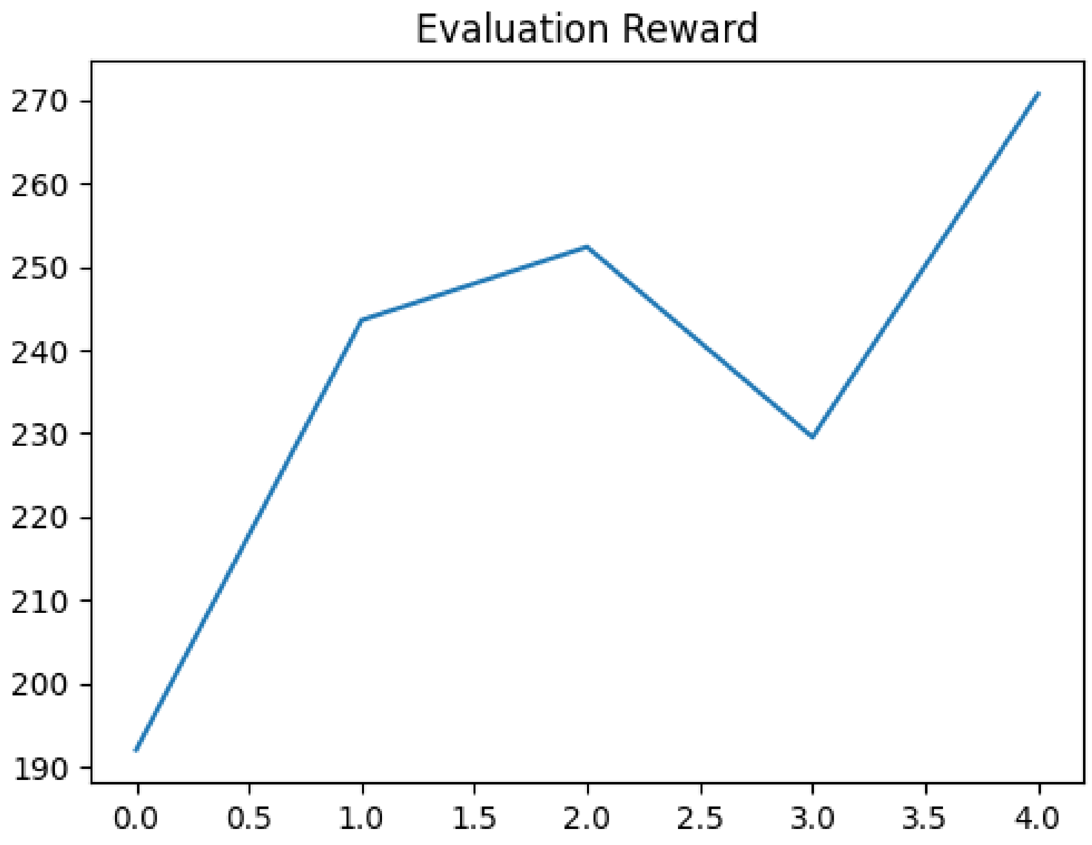


Figure 2: Training loss (left) and average evaluation reward (right) across generations.

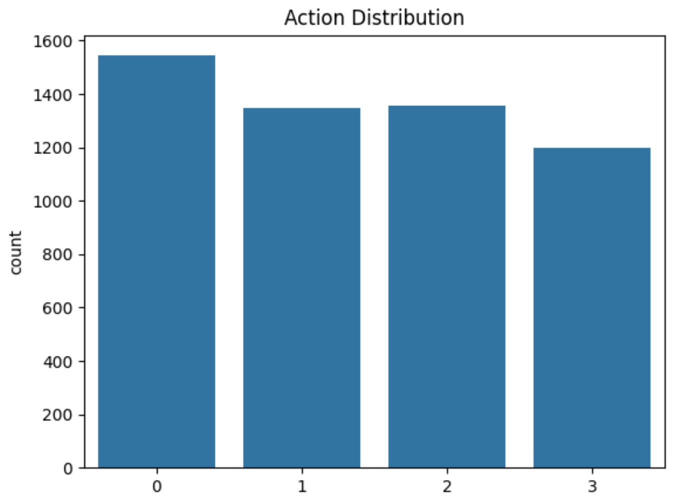


Figure 3: Distribution of actions selected by MCTS during self-play. Peaks indicate high-confidence decisions.

Conclusion

This implementation demonstrates the feasibility of combining MCTS and self-play in a single-agent control task. It shows that search-based planning can effectively bootstrap policy learning even without a traditional reinforcement learning signal, highlighting a direction for more stable, planning-driven RL architectures.

Conclusion

We implemented a multi-stage pipeline combining supervised imitation and iterative reinforcement learning to solve `LunarLander-v3`. Key insights:

- **Expert DQN** established a strong baseline (246.09 ± 83.56).
- **Behavioral Cloning** matched expert performance in mean reward but with high variance (246.72 ± 69.29).
- **DAgger** outperformed all methods (260.92 ± 34.09) by reducing compounding errors via expert feedback.
- **Self-Play** underperformed (157.73 ± 71.91), suggesting that pure on-policy refinement without expert guidance may require additional stabilization.

This study demonstrates that expert-informed imitation combined with targeted RL enhancements yields the most robust and high-performing agents. Future work will explore curriculum learning, prioritized replay, and distributional RL to further improve stability and performance in complex continuous environments.

Discussion

Challenges Encountered

Several challenges were faced during the course of this project, especially in training and optimizing the models. Some of the key challenges included:

- **Instability in Self-Play:** Although Self-Play achieved the best performance, it was initially unstable. The agent faced difficulties in exploring its environment effectively, especially early on in training when the Q-values were not yet fully refined.
- **Distributional Shift in Behavioral Cloning:** Behavioral Cloning struggled to generalize from the expert data, as the policy learned from offline demonstrations was unable to handle out-of-distribution states encountered during testing. This resulted in lower reward performance compared to the Expert DQN.

- **Computational Resource Constraints:** Training the models, particularly Self-Play and DAgger, required substantial computational resources, as multiple episodes and a large memory buffer were needed to stabilize the learning process. The extended training times for each iteration were another hurdle.

Sample Efficiency

We examined the sample efficiency of each method by measuring performance relative to the number of environment interactions:
DAgger and Enhanced Self-Play show competitive performance with relatively few interactions. Behavioral Cloning, though limited by lack of exploration, achieves decent results from just demonstration data. Enhanced Self-Play offers strong performance with reduced variance, suggesting improved policy stability.

Robustness Analysis

To evaluate robustness, we tested each model under perturbed initial conditions:
Standard Deviation of Rewards:

- Expert DQN: 106.18
- Behavioral Cloning: 111.90
- DAgger: 92.89
- Enhanced Self-Play: 44.79

Enhanced Self-Play demonstrates the most robust performance with the lowest reward variance and strong success rate. While DAgger maintains high performance, the reduced variance in Enhanced Self-Play suggests greater policy consistency and resilience to environmental noise.

Future Work

Future improvements could focus on several avenues:

- **Incorporating Curriculum Learning:** To address exploration inefficiencies in Self-Play, curriculum learning can be integrated. Starting the agent with simpler sub-tasks and gradually increasing difficulty may help stabilize learning.
- **Prioritized Experience Replay:** Implementing prioritized experience replay could improve data efficiency in Self-Play and DAgger, ensuring that important transitions are replayed more frequently, thus speeding up convergence.
- **Distributional Reinforcement Learning:** To better address the challenges of value estimation in continuous environments, exploring distributional RL methods like C51 could provide more robust Q-value approximations and better exploration.
- **Multi-Agent Scenarios:** Extending the Self-Play framework to multi-agent environments where agents can compete or cooperate would provide even more opportunities for policy improvement and exploration.