CLEAR-Net – Cart-pole Learning with Enhanced Adaptive Reinforcement Network

Aravind Narayanan
Department of Electrical and Computer Engineering
University of Toronto
`aravind.narayanan@mail.utoronto.ca`

Qilong Cheng
Department of Electrical and Computer Engineering
University of Toronto
`qilong.cheng@mail.utoronto.ca`

Abstract

This progress report presents the progress made in the project aimed at solving the Cart-Pole challenge using reinforcement learning (RL) with noisy observations. The objective is to develop and compare traditional non-deep RL algorithms and deep RL algorithms to evaluate their performance under sensor noise. The initial implementation of Q-learning and DQN is investigated and the results are discussed in this report.

Assentation of Teamwork

The work is contributed equally by both students on the project.

1 Introduction

The Cart-Pole problem is a classic control problem with two degree of freedom system and one actuator. The cart is able to move linearly on a rod, while the attached inverted pendulum on the cart rotates about the link on the cart. The actuation, the cart, is able to move either left or right on the rod. And the goal is to balance the inverted pendulum on the cart within the given length of the rod. Normally, the state space, position of the cart, velocity of the cart, angle of the pendulum and the angular velocity of the pendulum are given accurately, but in this project, Gaussian noise is assumed to be added to all the observation spaces. This project aims to solve this problem using reinforcement learning (RL) algorithms. We will be exploring both the traditional tabular Q-learning method of reinforcement learning, and different deep reinforcement learning methods to design the controller.

2 Environment Setup

The Cart-Pole problem can be formulated as an RL problem with the following components (1):

- **Agent:** The cart
- **Environment:** The Cart-Pole system provided by the Gymnasium library.
- **State:** The state is a four-dimensional vector consisting of:
  - Cart position (x)
  - Cart velocity ($\dot{x}$)
  - Pole angle (θ)
  - Pole angular velocity ($\dot{\theta}$)
- **Action:** Discrete actions of moving the cart left (0) or right (1).
- **Reward:** A reward of +1 for every time step the pole remains balanced.
- **Objective:** Maximize the cumulative reward over time.

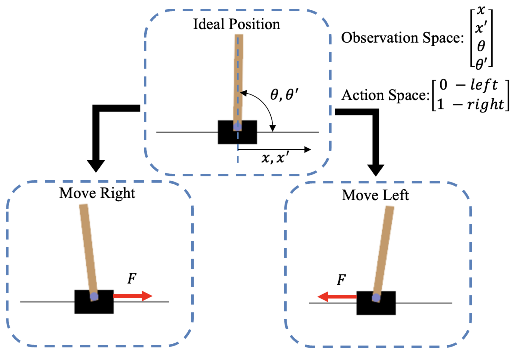

Ideal Position
θ, θ'
x, x'
Observation Space: $\begin{bmatrix} x \\ x' \\ \theta \\ \theta' \end{bmatrix}$
Action Space: $\begin{bmatrix} 0 - left \\ 1 - right \end{bmatrix}$
Move Right
F
Move Left
F

Figure 1: The overview of the environment and agent

In addition, it is worth mentioning the terminal state of the system defined by the environment. Namely, the cart position and the pole's angle are restricted. If the observations are out of those boundaries, the episode will terminate and end the game.

Table 1: Cart-Pole State and Observation Space

State Variable	Range	Description
Cart Position (x)	$[-4.8, 4.8]$	Position of the cart on the track
Cart Velocity ($\dot{x}$)	$(-\infty, \infty)$	Velocity of the cart
Pole Angle (θ)	$[-24°, 24°]$	Angle of the pole from the vertical
Pole Angular Velocity ($\dot{\theta}$)	$(-\infty, \infty)$	Angular velocity of the pole

Termination Conditions: The episode terminates when one of the following conditions is met:

- The pole angle (θ) is greater than 12 degrees from vertical.
- The cart position (x) is greater than 2.4 units from the center.
- The episode length reaches 500 time steps.

Observation Space: The observation space is continuous and consists of the four state variables: cart position, cart velocity, pole angle, and pole angular velocity.

3 Adding Noise to Observations

To simulate real-world sensor inaccuracies, Gaussian noise is added to each component of the observation vector. This is achieved using a custom Gymnasium Observation Wrapper that perturbs each state variable with zero-mean Gaussian noise with a configurable standard deviation (σ). In our initial testings, we picked $\sigma = 0.1$ as our testing sample.

In this implementation:

- The `NoisyObservationWrapper` class, inheriting from `gym.ObservationWrapper`, modifies environment observations.
- The `obs_ranges` array defines noise ranges:
  - Cart position: ± 2.4 units.
  - Cart velocity: ± 10 units.
  - Pole angle: ± 12 degrees.
  - Pole angular velocity: ± 10 degrees/second.
- The `observation` method introduces noise into the observation vector, scaled according to predefined ranges. It supports three noise models:
  - **Gaussian Noise (default)**: Normally distributed noise.
  - **Uniform Noise**: Noise uniformly distributed across a range.
  - **Exponential Noise**: Noise following an exponential distribution.

Below is a graph of the observation data with noise versus the ground truth over one episode with a random policy.

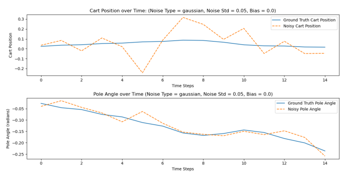


Figure 2: Gaussian noise observation results vs the ground truth results

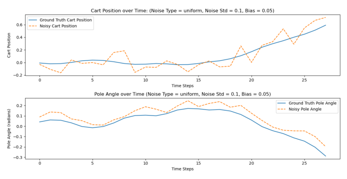


Figure 3: Uniform noise observation results vs the ground truth results with bias

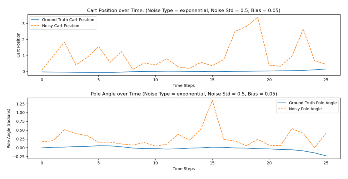


Figure 4: Exponential noise observation results vs the ground truth results

Measuring and Tuning the Noise Level: Adjust the standard deviation of Gaussian noise to control noise levels. Increasing the standard deviation adds more noise, while decreasing it reduces noise. Modify the noise range to test system robustness.

Future Improvements: Experiment with different standard deviations and noise ranges to improve the noise model and test robustness. If time permits, explore other noise distributions, such as those with bias.

4 Implement a Non-Deep RL Agent

First, we implemented the tabular approach to reinforcement learning, specifically Q-learning to estimate the action-state values of the system. Here is the pseudocode for the Q-learning algorithm:

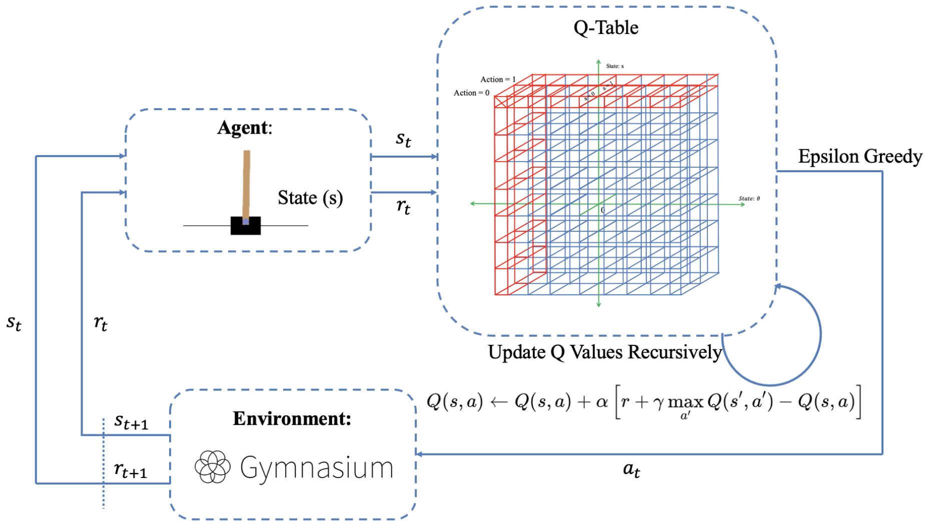


Figure 5: Classical Q-learning architecture overview

Algorithm 1 Q-Learning Algorithm

```
 1: Initialize Q-values arbitrarily for all state-action pairs
 2: for each episode do
 3:     Initialize state $s$
 4:     while state $s$ is not terminal do
 5:         Choose action $a$ from state $s$ using $\epsilon$-greedy policy
 6:         Take action aa, observe reward rr and next state ss'
 7:         Update Q-value: $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$
 8:         $s \leftarrow s'$
 9:     end while
10: end for=0
```

We trained the Q-learning agent for 100,000 episodes to ensure it had sufficient experience to learn an optimal policy. Initially, the agent explores randomly and gradually shifts to a greedy policy with fewer random actions. The state space is discretized into 8 or 12 bins per variable, which balances resolution and computational feasibility, providing high accuracy and efficiency compared to tests with 8 to 32 bins. Continuous observations are discretized by normalizing state variables within their ranges, scaling the normalized values to bin indices, and ensuring that the indices remain within valid limits.

Training Results

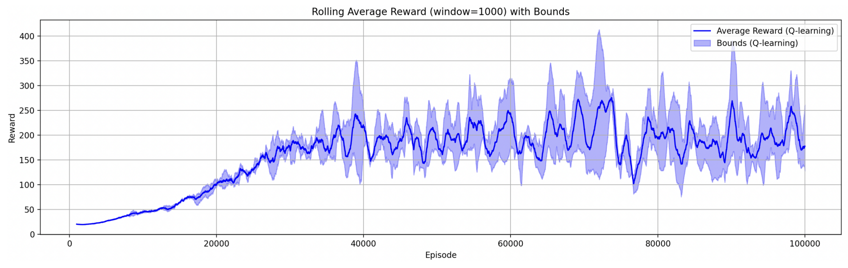


Figure 6: Q-learning training results over 100,000 episodes

Bin Size

Given that the size of discretization is crucial in classical reinforcement learning problems, we experimented with various discretization sizes for each observation state: cart position, cart velocity, pole angle, and pole angular velocity. We used discretization sizes of (8, 8, 8, 8), (8, 12, 8, 12), (16, 16, 16, 16), and (32, 32, 32, 32). Each configuration was trained over 100,000 episodes, and the results are shown in Figure 7.

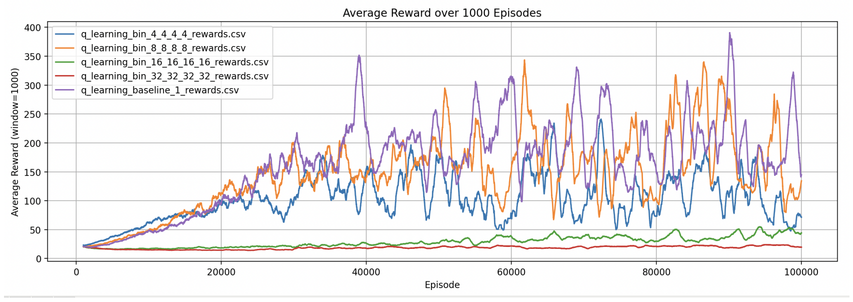


Figure 7: Comparison of average rewards over 100,000 episodes for various Q-learning configurations.

Bin Size (x, x', θ, θ')	**Runtime (seconds)**
4, 4, 4, 4	130.26
8, 8, 8, 8	187.47
8, 12, 8, 12	172.89
16, 16, 16, 16	36.37
32, 32, 32, 32	24.64

Table 2: Runtime for different bin sizes

Another observation can be made is that, due to the lack of convergence for the higher bin sized experiments, as can be referred from 2, the runtime is much lower compared to the lower bin sized model. However, bin size 4 performed poorly compared to the bin size of 8 or 12.

Noise Level

Finally, we trained the tabular approach using the baseline tested under different noise variances, namely with variance of 0.1, 0.3, 0.5 and 1. All models are trained over 500,000 episodes. The results show that Q-learning does not handle well given a higher noise level, and when the noise level reaches 0.5, the model already failed to converge to an optimal controller.

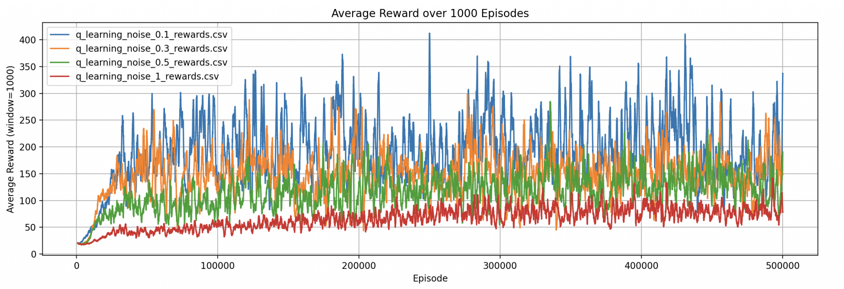


Figure 8: Q-Learning trained under different noise levels

During testing, we sampled 100 episodes, if the episode passes over 500 step, we will mark it as success. The results of different models trained under different noise level under different noise variance environment can be seen as follows:

Models	$\sigma = 0.1$	$\sigma = 0.3$	$\sigma = 0.5$	$\sigma = 1$
Trained under 0.1	81%[1]	21%	8%	0%
Trained under 0.3	53%	45%	10%	2%
Trained under 0.5	39%	36%	18%	1%
Trained under 1	23%	1%	5%	12%

Table 3: Models performance results under different noise levels using Q-Learning.

All models perform best at the noise level they were trained on. However, models trained on lower noise generally perform poorly when the environment noise increases. Similarly, models trained on higher noise do not perform as well when the noise decreases. In essence, high-noise-trained models excel under high-noise conditions. This may seem counterintuitive from a control perspective, but it could be due to the Q-value table memorizing actions and not adapting well when noise levels change. Finally, the model trained under 0.1 noise variance perform the best in all noise levels compared to other models. This could be due to its faster convergence compared to the other models.

Discussion

Classical RL algorithms like Q-Learning can solve the Cart-Pole problem but face challenges with continuous state spaces:

- **State Space Explosion:** Discretizing the state space exponentially increases state-action pairs, making computation infeasible. Using 20 bins is insufficient for high-resolution control.
- **Curse of Dimensionality:** Higher state space dimensions exponentially increase the number of discretized states. Increasing resolution by one results in a fourfold increase in state space dimension and a 4^4 increase in state-action values.
- **Approximation Errors:** Discretization introduces errors that limit performance, regardless of resolution.

Due to these complexities, classical RL is unsuitable for this project. Instead, deep RL algorithms, which use function approximation to handle continuous state spaces, will be explored next.

[1]The testing case percentage is calculated as the ratio of the number of successful trials to 100 random trials under each noise level. Note that all the percentages in this report are tested under 100 trails

5 Deep Q-Learning (DQL)

Deep Q-Learning (DQN) leverages neural networks to approximate the Q-values for continuous state spaces. It uses experience replay and a target network to stabilize training. The key components are:

- **Experience Replay:** Stores past experiences and samples mini-batches for training the network.
- **Target Network:** A separate network for stable Q-value targets, updated less frequently than the main network.

Algorithm 2 Deep Q-Learning (DQN) Algorithm

```
1: Initialize replay memory $D$ and action-value function $Q$ with random weights
2: for each episode do
3:     Initialize state $s$
4:     while state $s$ is not terminal do
5:         Choose action $a$ from state $s$ using $\epsilon$-greedy policy based on $Q$
6:         Take action $a$, observe reward $r$ and next state $s'$
7:         Store transition $(s, a, r, s')$ in replay memory $D$
8:         Sample random mini-batch of transitions $(s_j, a_j, r_j, s_{j+1})$ from $D$
9:         Set target $y_j = r_j + \gamma \max_{a'} Q(s_{j+1}, a')$ if $s_{j+1}$ is not terminal, otherwise $y_j = r_j$
10:        Perform gradient descent step on $(y_j - Q(s_j, a_j))^2$
11:        $s \leftarrow s'$
12:    end while
13:    Update target network $Q'$ with weights of $Q$ every $C$ steps
14: end for=0
```

The neural network used in the DQN Algorithm is as follows:

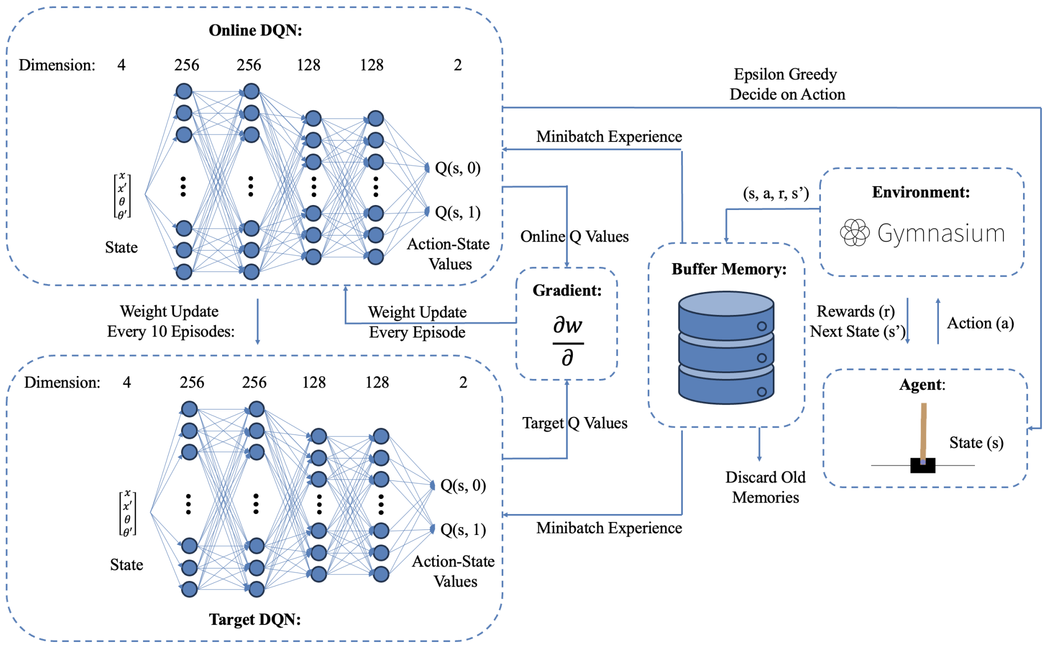


Figure 9: DQL architecture Overview

We were able to get the preliminary results done and an initial neural network built using CNN. Few things we noticed instantly during our experiments are:

- **Training time:** The training time drastically reduced compared to the Q-learning method. Although for each episode it took longer for the training to take place, overall, DQN is faster to converge to an ideal solution.

- **Number of Episodes:** This drastically reduced too from the previous 100000 to now 10000.

- **Controller Performance:** The testing reward from using the DQN method is a long higher than Q-learning method.

- **Runtime:** However, the training runtime for DQL is much higher than that of Classical Q-learning approach.

5.1 Hyperparameter Tuning

First, we investigated the effects of different hyper-parameters on the performance of the model. Specifically, the learning rate, batch size, and epsilon decay rate. More details and results can be referred in the **Appendix**.

To sum up the optimal hyper-parameters are as follows:

Hyperparameters	**Values**
Learning Rate	0.0001
Batch Size	64
Epsilon Decay Rate	0.9995
Episodes	5000
Target Network Update Rate	10
Terminal Step Size	500

Table 4: Hyperparameters chosen for the baseline DQL network

5.2 Reward Shaping

In our experiment, the cart-pole agent achieved high rewards but failed to stay centered, exhibiting drifts and deviations. To address this, we experimented with reward shaping.
Reward Function 1:

$$\text{reward\_fun1}(x, \dot{x}, \theta, \dot{\theta}, R, d) = \begin{cases} 1.0 - \frac{|x|}{2.4} - \frac{|\theta|}{0.209} - \frac{|\dot{x}|}{1.0} - \frac{|\dot{\theta}|}{1.0} & \text{if } R \geq 500 \text{ or not } d, \\ -0.1 - \frac{|x|}{2.4} - \frac{|\theta|}{0.209} - \frac{|\dot{x}|}{1.0} - \frac{|\dot{\theta}|}{1.0} & \text{if } R < 500 \text{ and } d. \end{cases}$$

This function rewards the agent for maintaining the cart near the center and the pole upright, with penalties for early termination if total rewards are below 500. **Reward Function 2:**

$$\text{reward\_fun2}(x, \dot{x}, \theta, \dot{\theta}, R, d) = \begin{cases} 1.0 & \text{if } |x| < 0.5 \text{ and } |\theta| < 0.05, \\ -\frac{|x|}{2.4} - \frac{|\theta|}{0.209} & \text{otherwise,} \\ -1.0 & \text{if terminated and } R < 500. \end{cases}$$

This function gives a significant reward for small cart positions and pole angles, penalizes deviations, and applies a larger penalty for early termination if total rewards are below 500.

Both functions aim to stabilize the cart-pole system by rewarding balanced states and penalizing deviations. And the training results are shown below. During testing reward function does help the agent stay in the middle of the track, but it sacrificed the controller performance, as it degrades from the baseline performance. In the future work, other reward functions could be tested to achieve better convergence speed and performance.

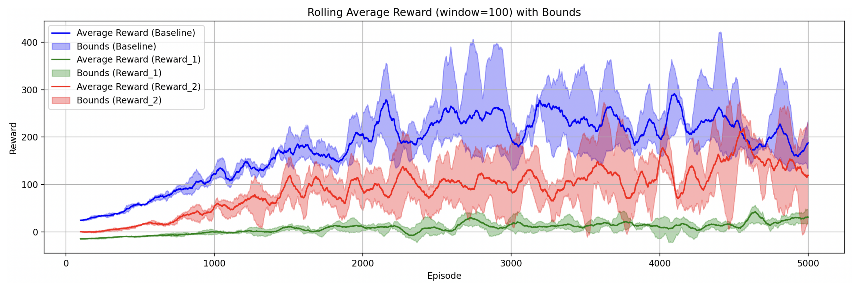


Figure 10: Training results based on different reward functions

Reward Function	**Accuracy**
Baseline	100% [2]
Reward Function 1	7%
Reward Function 2	88%

Table 5: Agents trained under different noise level(different standard derivations) performance

5.3 Effects on noise level

Next, we experimented with the noise level. The initial experiments listed above were all trained under Gaussian noise with a standard derivation of 0.1. We are particularly interested in two questions:

- Does the model trained under 0.1 standard derivation perform well under higher level of noise variance?
- Does the model trained under a higher variance of noise perform better than the one with less variance?

The training results of the models under different noise variance are shown in Figure 11. After training, all models are tested under different level of observation noise, and the results are as follows:

Models	$\sigma = 0.1$	$\sigma = 0.3$	$\sigma = 0.5$	$\sigma = 1$	$\sigma = 2$
Trained under 0.1	100% [2]	100%	84%	0%	15%
Trained under 0.3	78%	100%	78%	2%	33%
Trained under 0.5	0%	99%	91%	6%	39%
Trained under 1	0%	46%	48%	5%	12%
Trained under 2	0%	0%	0%	0%	1%

Table 6: Different model performance trained under different level of noise under different noise level

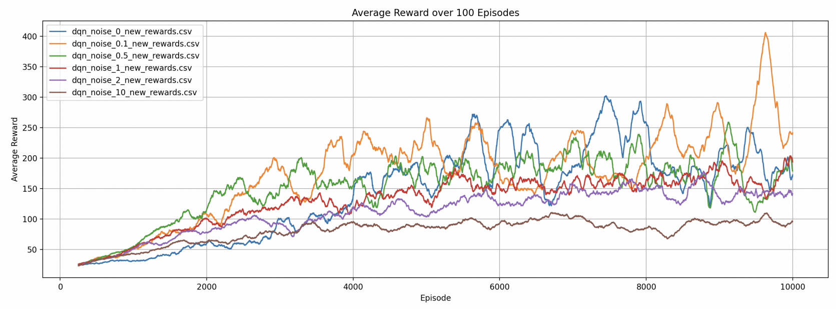


Figure 11: Effect of the noise level on the performance of DQL models

[2]The testing case percentage is calculated as the ratio of the number of successful trials to 100 random trials under each noise level. Note that all the percentages in this report are tested under 100 trails

The results yielded several noteworthy observations:

- Models generally perform best at the noise level they were trained on, although this is not always true for models trained under very high noise levels (e.g., $\sigma = 1$ and $\sigma = 2$).
- Models trained at low noise levels struggle to handle higher noise, whereas models trained at high noise levels perform better under high noise conditions but worse under low noise conditions.

Interestingly, a model trained with a noise variance of 0.5 outperformed those trained with lower noise when evaluated at 0.5 noise variance. Similarly, a model trained with a variance of 2 succeeded in balancing the pole at that noise level, where other models failed, but performed poorly at a noise level of 0.1.

Future research should explore developing models capable of handling a wide range of noise levels. One potential approach is to vary the noise level during training, allowing the model to adapt and balance the agent effectively across different noise environments.

5.4 Disturbances

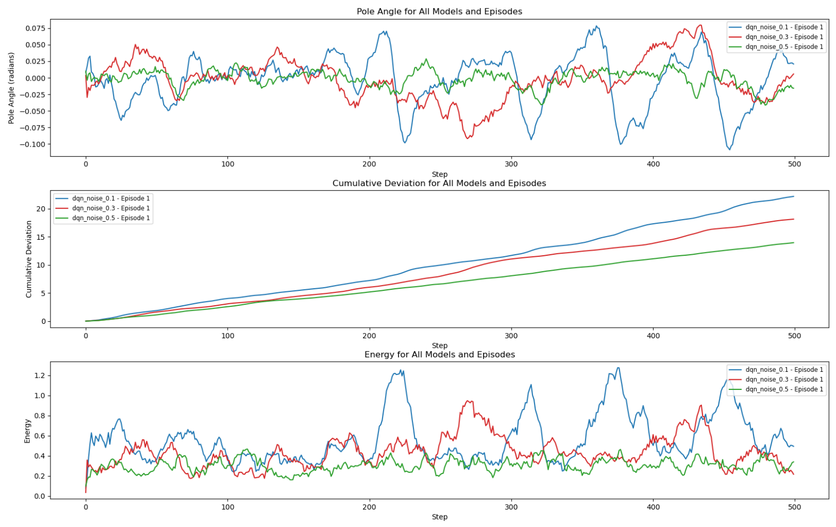


Figure 12: Robustness comparison of DQN models with varying noise levels under identical disturbances

Next we investigated implusive disturbance's effect on different models and how is each controller's robustness perform.
In this experiment we applied disturbances manually to the pole angle to assess the robustness of different Deep Q-Network (DQN) controllers. We tested three DQN models trained with noise levels of 0.1, 0.3, and 0.5, and recorded pole angle deviations, cumulative deviations from upright, and system energy over time.

- **Pole Angle Variations**: The model trained with lower noise (0.1) exhibited more pronounced angle fluctuations, suggesting it struggles with disturbances. The model trained with higher noise (0.5) showed smoother variations, indicating better adaptability.
- **Cumulative Deviation**: The model trained with the highest noise level (0.5) had the lowest cumulative deviation, reflecting superior long-term stability in maintaining the pole near upright.
- **System Energy**: Energy spikes in the lower-noise models indicate less stability, while the high-noise model managed system dynamics more effectively, with a smoother energy profile under perturbations.

Overall, training with higher noise levels enhances controller robustness, making the system more resilient to real-world disturbances.

6 Policy Gradient Method (PGM)

After implementing the DQL approach, we decided to experiment with the PPO algorithm due to its recent popularity and effectiveness. Although the environment is defined with a discrete action space, the classic control problem actually involves continuous force applied to the cart. Therefore, using a Policy Gradient approach is also appropriate in this context.

Algorithm 3 Proximal Policy Optimization (PPO)

```
0: while not converged do
0:   Initialize empty trajectory buffer $\mathcal{D}$
0:   for t = 1, 2, ..., T do
0:     Sample action $a_t \sim \pi_\theta(a_t|s_t)$
0:     Execute action $a_t$ in the environment
0:     Observe reward $r_t$ and next state $s_{t+1}$
0:     Store $(s_t, a_t, r_t, s_{t+1})$ in $\mathcal{D}$
0:   end for
0:   Compute advantages $\hat{A}_t$ using $V_\phi(s_t)$ and rewards $r_t$
0:   for epoch = 1, 2, ..., K do
0:     for batch $b \in \mathcal{D}$ do
0:       Compute the ratio $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$
0:       Compute the clipped surrogate objective:
0:       $L^{CLIP}(\theta) = \mathbb{E}_t\left[\min\left(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t\right)\right]$
0:       Update the policy network by maximizing $L^{CLIP}(\theta)$
0:       Update the value function by minimizing the loss $L^{VF}(\phi) = \mathbb{E}_t\left[(V_\phi(s_t) - R_t)^2\right]$
0:     end for
0:   end for
0: end while=0
```

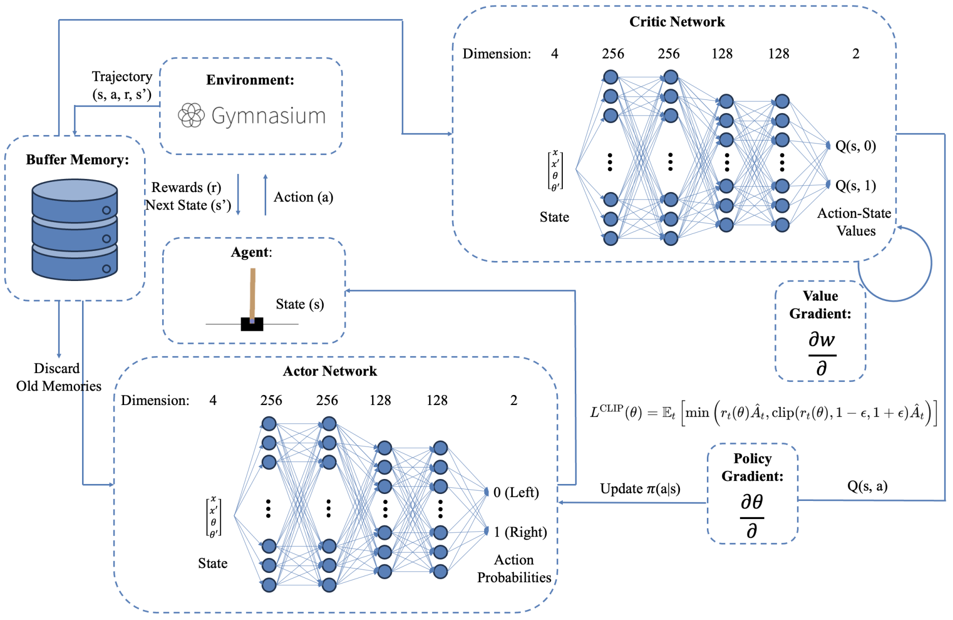

Critic Network
Dimension: 4 256 256 128 128 2
State
Q(s, 0)
Q(s, 1)
Action-State Values
Trajectory (s, a, r, s')
Environment:
Gymnasium
Buffer Memory:
Rewards (r) Next State (s')
Action (a)
Agent:
State (s)
Value Gradient:
$\frac{\partial w}{\partial}$
Discard Old Memories
Actor Network
Dimension: 4 256 256 128 128 2
State
0 (Left)
1 (Right)
Action Probabilities
$L^{\text{CLIP}}(\theta) = \mathbb{E}_t\left[\min\left(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t\right)\right]$
Policy Gradient:
$\frac{\partial \theta}{\partial}$
Update π(a|s)
Q(s, a)

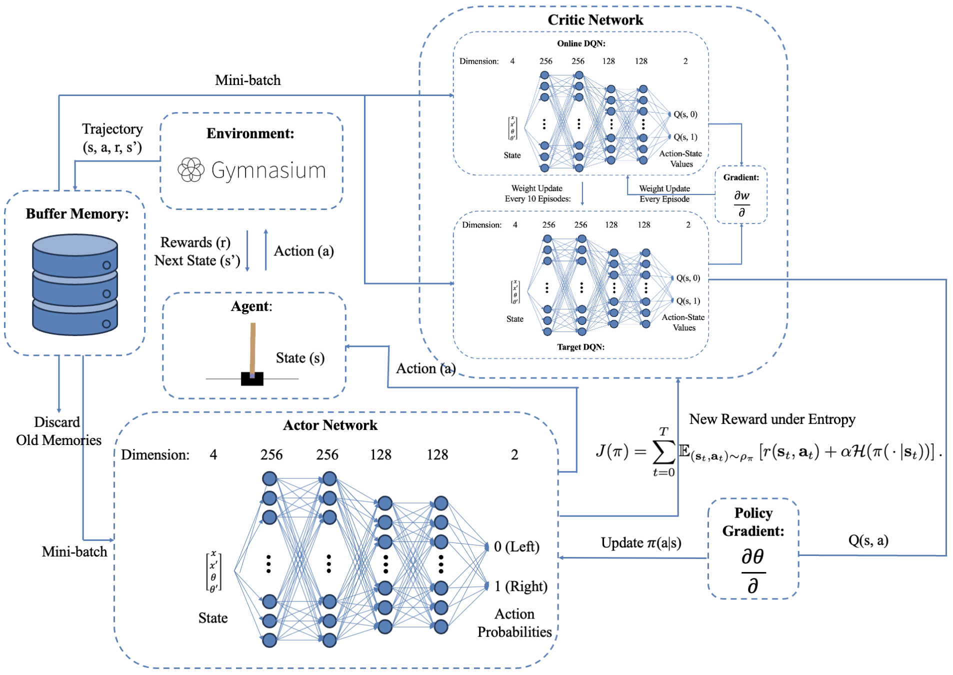
Figure 13: Proximal Policy Gradient architecture overview

7 Soft Actor-Critic (SAC)

Finally, to further improve the stability of the agent and experiment with different architectures, we implemented the Soft Actor-Critic (SAC) algorithm to the agent (2). SAC balances deterministic and stochastic policies using an additional entropy term for optimal exploration and exploitation and eliminated the need for tuning epsilon decay rate. It also uses twin Q-networks for increased stability and an action-space wrapper to convert discrete actions to continuous actions. This allows us to deploy SAC algorithm into the discrete action-space problem.

Algorithm 4 Soft Actor-Critic (SAC)

```
1:  Initialize policy network $\pi_\theta(a|s)$, Q-value networks $Q_{\phi_1}(s,a)$, $Q_{\phi_2}(s,a)$, value network $V_\psi(s)$,
    and target value network $V_{\psi'}$ with $\psi' \leftarrow \psi$
2:  Initialize replay buffer $\mathcal{D}$
3:  for each iteration do
4:      for each environment step do
5:          Sample action $a_t \sim \pi_\theta(a_t|s_t)$, execute $a_t$, observe $r_t$, $s_{t+1}$
6:          Store $(s_t, a_t, r_t, s_{t+1})$ in $\mathcal{D}$
7:      end for
8:      for each update step do
9:          Sample mini-batch from $\mathcal{D}$
10:         Compute target value $V_{\text{target}} = r_t + \gamma \mathbb{E}_{a_{t+1}\sim\pi_\theta}[V_{\psi'}(s_{t+1}) - \alpha \log \pi_\theta(a_{t+1}|s_{t+1})]$
11:         Update Q-values by minimizing $L_Q(\phi_i) = \mathbb{E}[(Q_{\phi_i}(s_t,a_t) - V_{\text{target}})^2]$ for $i = 1, 2$
12:         Update value network by minimizing $L_V(\psi) = \mathbb{E}[(V_\psi(s_t) - \mathbb{E}[Q_{\phi_1}(s_t,a_t) -$
            $\alpha \log \pi_\theta(a_t|s_t)])^2]$
13:         Update policy network by minimizing $L_\pi(\theta) = \mathbb{E}[\alpha \log \pi_\theta(a_t|s_t) - Q_{\phi_1}(s_t,a_t)]$
14:         Soft update target value network: $\psi' \leftarrow \tau\psi + (1-\tau)\psi'$
15:     end for
16: end for=0
```



Critic Network
Online DQN:
Dimension: 4 256 256 128 128 2
State
Q(s, 0)
Q(s, 1)
Action-State Values
Weight Update Every 10 Episodes:
Weight Update Every Episode
Gradient:
$\frac{\partial w}{\partial}$
Target DQN:
Mini-batch
Trajectory (s, a, r, s')
Environment:
Gymnasium
Buffer Memory:
Rewards (r) Next State (s')
Action (a)
Agent:
State (s)
Action (a)
Discard Old Memories
Actor Network
Dimension: 4 256 256 128 128 2
State
0 (Left)
1 (Right)
Action Probabilities
Mini-batch
New Reward under Entropy
$J(\pi) = \sum_{t=0}^{T} \mathbb{E}_{(\mathbf{s}_t,\mathbf{a}_t)\sim\rho_\pi}[r(\mathbf{s}_t,\mathbf{a}_t) + \alpha\mathcal{H}(\pi(\cdot|\mathbf{s}_t))]$.
Update π(a|s)
Policy Gradient:
$\frac{\partial\theta}{\partial}$
Q(s, a)

Figure 14: Soft Actor-Critic architecture overview

Further, since SAC converged under 200 episodes, we trained the network more rigorously with a higher noise variance and resulted with the following.

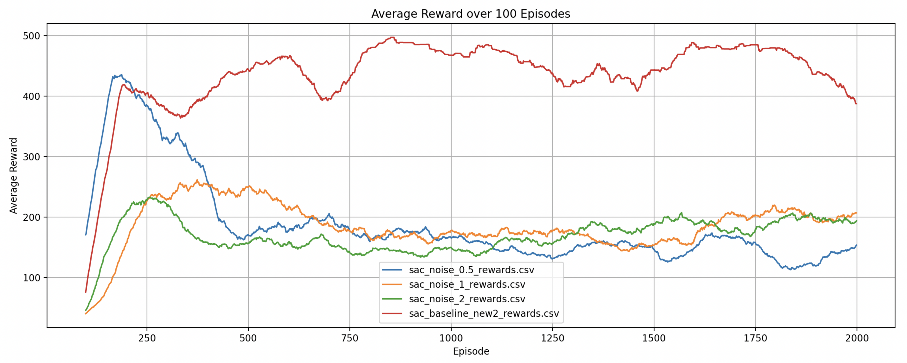


Figure 15: Different noise level's effect on the training of SAC algorithm

It can be seen that the downside of SAC algorithm is that it does not generalize well under higher noise variance. It converges at the beginning of the first 200 episodes but quickly forgets. However, it does stabilize well after few hundred more iterations of training.

8 Comparison of Different DRL

The most noticeable difference is that Deep Reinforcement Learning (DRL) algorithms require significantly fewer episodes to converge compared to the Q-learning approach. Notably, Soft Actor-Critic (SAC) converges the fastest due to its elimination of the epsilon term, leading to enhanced exploration and consistent performance. Deep Q-Learning (DQL) converges faster than Proximal Policy Optimization (PPO) but tends to suffer from "catastrophic forgetting," causing instability. While all learning-based algorithms are susceptible to overfitting, resulting in some degree of "forgetting," PPO is the most stable and robust over extended training periods. However, DRL take longer to train compared to the tabular RL due to the higher computational efforts.

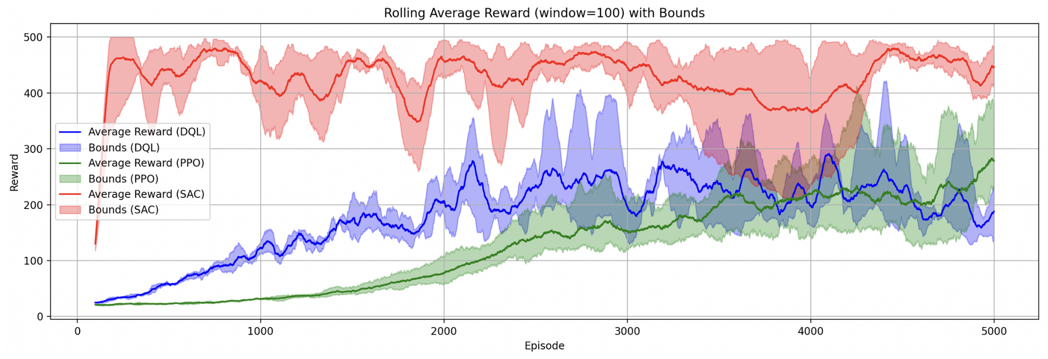


Figure 16: Training results comparison between three different deep reinforcement learning methods

9 Conclusion

Our experiments show that classical reinforcement learning struggles with continuous observation spaces due to the exponential growth of state-action pairs and required training time. In contrast, deep learning significantly reduces the number of required episodes and improves performance. Although deep reinforcement learning generally requires more training time and can suffer from catastrophic forgetting, PPO and SAC exhibit superior stability and robustness, making them suitable for real-world dynamic environments. Hyperparameter tuning and reward shaping are essential for enhancing performance and reducing training episodes. Additionally, models trained with higher noise variance perform better under sudden disturbances, achieving faster stability and lower energy requirements.

References

[1] Gymnasium, “Cart-Pole Environment,” https://gymnasium.farama.org/environments/classic_control/cart_pole/, accessed July 1, 2024.

[2] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, “Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor,” in *Proceedings of the 35th International Conference on Machine Learning*, 2018, pp. 1861–1870.

Appendix

Learning Rate Effects

The best performed model has the learning rate of 0.0001. The smaller learning rate have a hard to converge to an optimal solution, while when learning rate is 0.00001, it suffers from catastrophic forgetting more severely than the others.

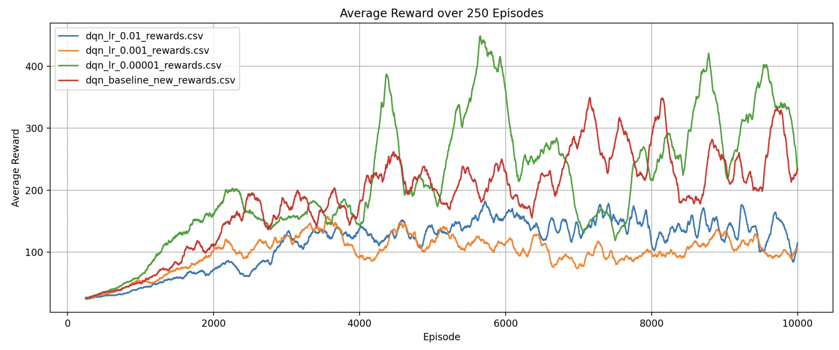


Figure 17: Training results based on different learning rate. Baseline is chosen with 0.0001 learning rate

Learning Rate	Accuracy
0.01	0%
0.001	0%
0.0001	100%
0.00001	87%

Table 7: Learning rate effects comparison on the model’s performance using DQL

Batch Size Effects

Three different batch sizes have been tested and we have chosen the batch size of 64 as our baseline parameter.

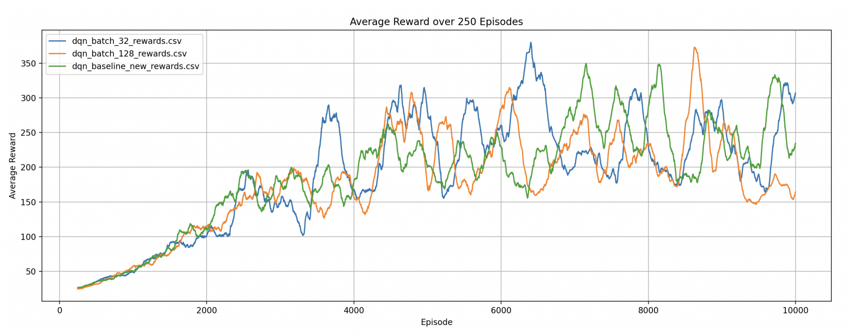


Figure 18: Training results based on different batch size. Baseline is chosen with batch size of 64

Batch Size	**Accuracy**
32	0%
64	44%
128	30%

Table 8: Batch size effects on the model's performance using DQL

Decay Rate Effects

Finally different epsilon decay rate has been tested. There are few interesting observations:

- The smaller the epsilon decay is, the faster the network can be converged. Even when $\epsilon = 0.99$, the network still stabilized, but did suffer from over-fitting

- Even though with a higher epsilon decay rate, the network converge slower, it tends to be more stable in the later training.

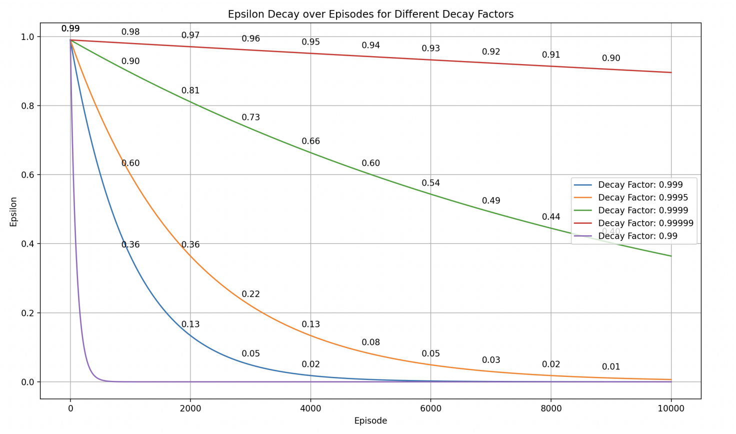


Figure 19: Different epsilon decay rate vs number of episodes

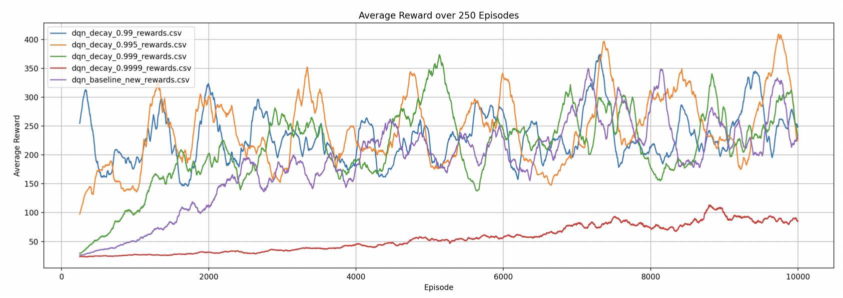


Figure 20: Training results based on different epsilon decay rate. Baseline is chosen with an epsilon decay rate of 0.9995

Epsilon Decay Rate	Number of Episodes to Converge
0.99	100%
0.995	92%
0.999	99%
0.9995	96%
0.9999	18%

Table 9: Epsilon decay's effects on the model's performance using DQL