Deep Rrinforcement Learning

Anndischeh Mostafavi- 220051723

Course: INM707 – 2024–2025

GitHub Link

W&B Link

Typeset using LaTeX

Basic

1 Define an environment and the problem to be solved

Environment: The environment, shown in figure1 , is a discrete 2D grid world of size 5x5, implemented using the *Grid-World* class. The agent can move in four directions: up, down, left, and right. The environment includes a start state $(0, 0)$, a goal state $(4, 4)$, and obstacles located at $(1, 1)$ and (2, 3). The environment also limits the maximum steps to 100. **Problem:** The

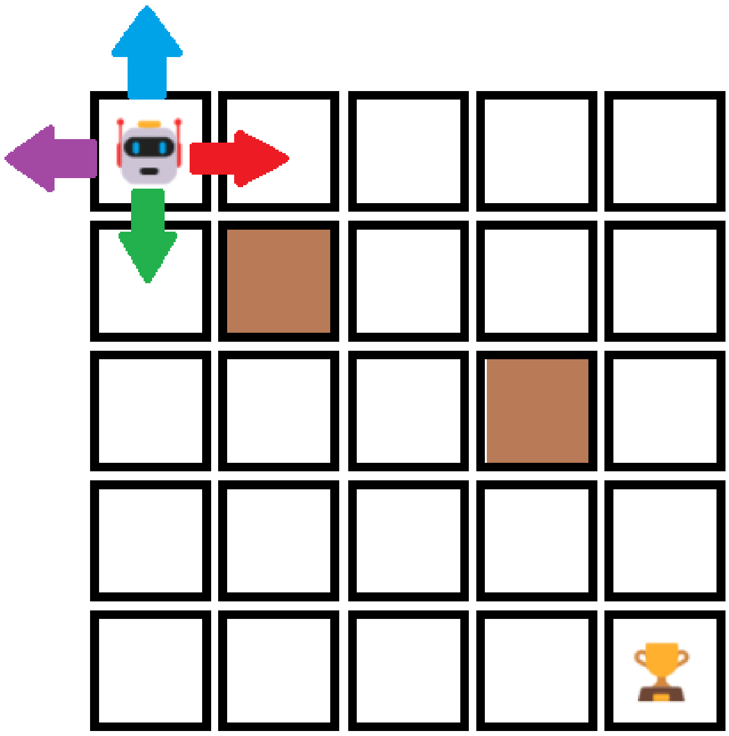

Fig. 1: Basic agent navigating towards the goal.

agent must learn to navigate the grid world from the start state to the goal state while avoiding obstacles. The agent receives rewards based on its actions and the resulting states. The problem is solved when the agent consistently reaches the goal state efficiently, demonstrating a learned optimal policy.

2 Define a state transition function and the reward function

State Transition Function - $P(s' \mid s, a)$: In this specific grid world environment, the state transition function [1], $P(s' \mid s, a)$, is largely deterministic, meaning that given a state s and an action a, the next state s' is almost always predictable. However, the environment also incorporates constraints (grid boundaries, obstacles) that influence the actual transition.

$$P(s' \mid s, a) = \begin{cases} 1 & \text{if } s' = f(s, a) \text{ and } s' \notin \text{Obstacles and } s' \in \text{Grid}, \\ 1 & \text{if } s' = s \text{ and } (f(s, a) \notin \text{Grid or } f(s, a) \in \text{Obstacles}), \\ 0 & \text{otherwise.} \end{cases} \tag{1}$$

Where:

- ▷ s represents the current state, a tuple (row, col) representing coordinates on the grid.
- ▷ a represents the action taken, belonging to the action space $\{0, 1, 2, 3\}$ (Up, Down, Left, Right).
- ▷ s' represents the next state.
- ▷ $f(s, a)$ is a function that calculates the intended next state based on the action a taken in state s. It represents the movement dynamics.

- If $a = 0$ (Up), $f(s, a) = (\text{row} - 1, \text{col})$
- If $a = 1$ (Down), $f(s, a) = (\text{row} + 1, \text{col})$
- If $a = 2$ (Left), $f(s, a) = (\text{row}, \text{col} - 1)$
- If $a = 3$ (Right), $f(s, a) = (\text{row}, \text{col} + 1)$

▷ $s' \in$ Grid means the state s' is within the bounds of the Grid. In other words, the row and column are within $[0, \text{grid\_size} - 1]$.

▷ $s' \notin$ Obstacles means s' is not inside any obstacle in the GridWorld.

▷ $f(s, a) \notin$ Grid or $f(s, a) \in$ Obstacles refers to the cases when the agent will stay in the same place. This happens if the agent's next movement because of action a makes the agent go off-grid or the agent will be in an obstacle.

Reward Function - $R(s, a, s')$: The reward function $R(s, a, s')$ provides feedback to the agent after each transition. It quantifies the desirability of transitioning from state s to state s' after taking action a.

$$R(s, a, s') = \begin{cases} R_{\text{goal}} & \text{if } s' = \text{GoalState}, \\ R_{\text{obstacle}} & \text{if } s' \in \text{Obstacles}, \\ R_{\text{step}} & \text{otherwise.} \end{cases} \tag{2}$$

Where:

▷ R_{goal} represents the reward for reaching the goal state. In the provided code, $R_{\text{goal}} = 10$.

▷ R_{obstacle} represents the penalty for entering an obstacle state. In the provided code, $R_{\text{obstacle}} = -10$.

▷ R_{step} represents the cost for taking each step. In the provided code, $R_{\text{step}} = -0.1$.

▷ GoalState is the coordinates of the goal states in the grid world.

▷ $s' \in$ Obstacles means the state s' is inside any obstacle in the GridWorld.

3 Set up the Q-learning parameters (gamma, alpha) and policy

Q-Learning Parameters and Convergence: Q-learning's convergence relies critically on the learning rate ($\alpha \in [0, 1]$) and discount factor ($\gamma \in [0, 1]$) [2]. As shown in equation 3, the Q-learning update rule defines the relationship between the learning rate, discount factor, and the Q-values :

$$Q(s, a) \leftarrow Q(s, a) + \alpha \left[R(s, a, s') + \gamma \max_{a'} Q(s', a') - Q(s, a) \right] \tag{3}$$

Demonstrates α as the weight applied to new information. A high α accelerates learning but risks instability due to high variance updates. Conversely, a low α ensures stable convergence but slows learning.
The parameter γ dictates the importance of future rewards. A γ close to 1 approximates the total discounted reward, promoting long-term planning, while a γ near 0 focuses solely on immediate rewards, resulting in a myopic policy.
Selecting appropriate α and γ is crucial for balancing bias and variance in the Q-value estimates and ensuring convergence to the optimal Q.
Epsilon-Greedy Policy and Exploration-Exploitation Tradeoff: The action selection strategy employs an epsilon-greedy policy:

$$a = \arg\max_a Q(s, a) \quad \text{with probability} \quad (1 - \epsilon) \quad \text{(exploitation)},$$

and

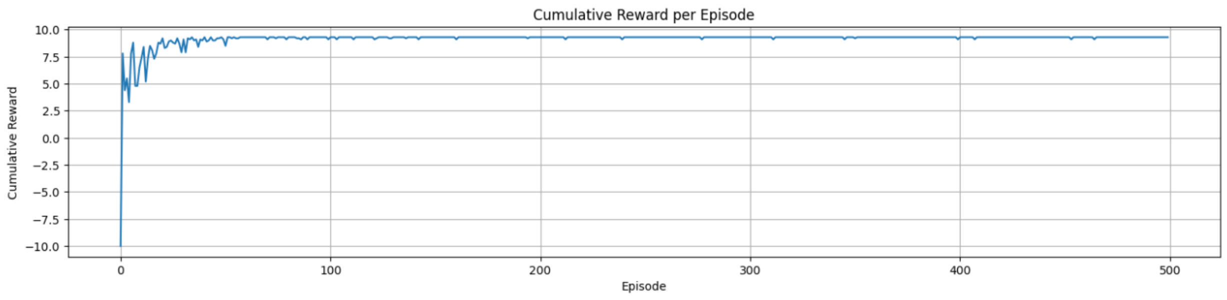

Fig. 2: Cumulative reward per episode during Q-learning training[3].

$$a = \text{random action} \quad \text{with probability} \quad \epsilon \in [0, 1] \quad \text{(exploration)}.$$

This balances the exploration-exploitation dilemma. A high ϵ promotes exploration, potentially escaping local optima but delaying convergence. A low ϵ favors exploitation, potentially converging to a suboptimal policy.
The ϵ-decay (e.g., $\epsilon \leftarrow$ epsilon-decay rate) gradually shifts the policy from exploration to exploitation as the agent learns. The initial ϵ, decay rate, and minimum ϵ are hyperparameters tuning the exploration schedule.

4 Run the Q-learning algorithm and represent its performance

Q-learning Algorithm Implementation: The Q-learning algorithm was implemented to train an agent to navigate the defined grid world environment. A single run was conducted as a baseline, with the algorithm configured as follows:

- ▷ Learning rate (α) = 0.1
- ▷ Discount factor (γ) = 0.9
- ▷ Exploration rate (ϵ) = 0.1
- ▷ Total training episodes = 1000
- ▷ ϵ decays with the decay rate of 0.99

The algorithm was executed using the `run-experiment` function, which in turn calls the core Q-learning function, iteratively updating the Q-table based on the agent's interactions with the environment. As part of each episode, the cumulative reward and episode length are tracked for subsequent performance analysis.
The performance of the Q-learning algorithm was visualized using two key metrics: cumulative reward per episode, figure 2, the plot illustrating cumulative reward provides insights into the agent's learning progress over time. Initially, rewards fluctuate significantly, reflecting the exploration phase where the agent randomly samples actions to understand its environment. As training progresses, the cumulative reward increases and plateaus, indicating the agent's growing ability to make decisions that lead to the goal state.
And episode length per episode, figure3. The second plot shows the episode length, defined as the number of steps taken to reach the goal. As the agent learns an efficient policy, the episode length decreases. The plot helps to show how long it take the agent to reach a state.

5 Repeat the experiment with different parameter values, and policies

To evaluate the sensitivity of the Q-learning algorithm to various hyperparameter settings, a parameter sweep was performed over different values of α, γ, and ϵ. The learning rate (α) was varied across 0.1 and 0.3, modulating the speed at which the Q-values were updated. The discount factor (γ) was adjusted to 0.7 and 0.9, impacting the agent's consideration of future rewards versus immediate ones. The exploration rate (ϵ) was set to 0.1 and 0.3, controlling the exploration-exploitation trade-off. The analysis included a full factorial design, testing all eight

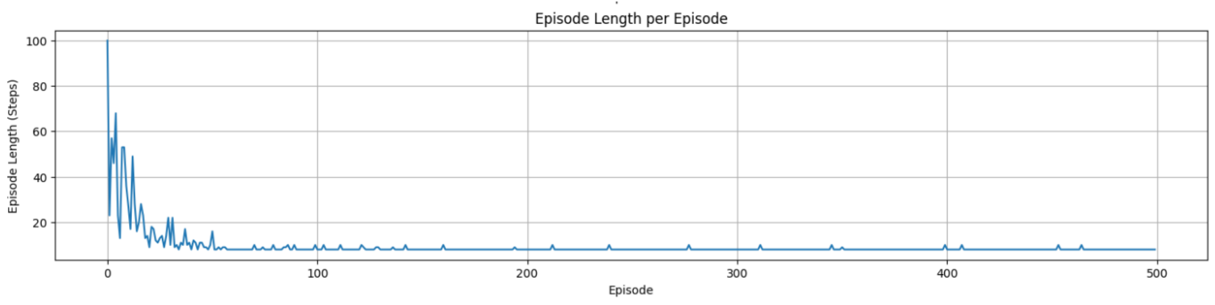

Fig. 3: Episode Length per Episode[3].

Tab. 1: Hyperparameter combinations and their average rewards[3].

Alpha	Gamma	Epsilon	Avg. Reward
0.1	0.7	0.1	9.283
0.1	**0.7**	**0.3**	**9.295**
0.1	0.9	0.1	9.283
0.1	0.9	0.3	9.295
0.3	0.7	0.1	9.290
0.3	0.7	0.3	9.292
0.3	0.9	0.1	9.292
0.3	0.9	0.3	9.290

Note: Bold value indicates the best performing parameter combination.

combinations of these parameters, and assessing how each unique setup affected the learning process. Each combination was trained for 500 episodes.
It is a useful technique as it allows us to view which parameters have the best policy in the GridWorld. As part of analyzing the best policy, plots were produced to show the cumulative reward and episode length over time. The best combination of α, γ, and ϵ would then be used in the GridWorld to analyze how well the model performs in an environment.

6 Analyze the results quantitatively and qualitatively

Quantitatively, the average reward, figure 2, over the last 100 episodes was used as the primary metric to compare the performance of different parameter settings. This metric captures the agent's ability to consistently achieve high rewards after the initial learning phase, ensuring that the agent consistently reaches the goal state with some degree of optimality. The results of the parameter sweep are summarized in Table 1.
As seen in Table 1, the best-performing parameter combination was $\alpha = 0.1$, $\gamma = 0.7$, and $\epsilon = 0.3$, achieving an average reward of 9.295 over the last 100 episodes. This combination indicates that a relatively low learning rate, prioritizing immediate rewards, and a moderate level of exploration resulted in the most effective learning.
Qualitatively, the policies learned by the different parameter settings were visualized using heatmaps. The heatmap in Figure 4 displays the learned policy corresponding to the best parameter settings identified quantitatively in Table 1. Qualitatively, the policy maps were examined to understand whether the learned policies aligned with intuitive expectations. An optimal policy is characterized by the agent consistently moving towards the goal state while avoiding obstacle states. For example, in the best-performing policy, the agent, starting from the initial state (0, 0), is directed right (action ID 3), then downwards, and then right again, which avoids the obstacle, and gets it to its goal. This is also characterized in other states where obstacles are present.
Therefore, the Figure 4 was analyzed to determine whether the parameters used produced a policy map which avoids the obstacles. It is important to evaluate whether the agent is consistently achieving the goals. Furthermore, the best parameter Figure 5 can be used to obtain insights on the exploration and exploitation tradeoff.

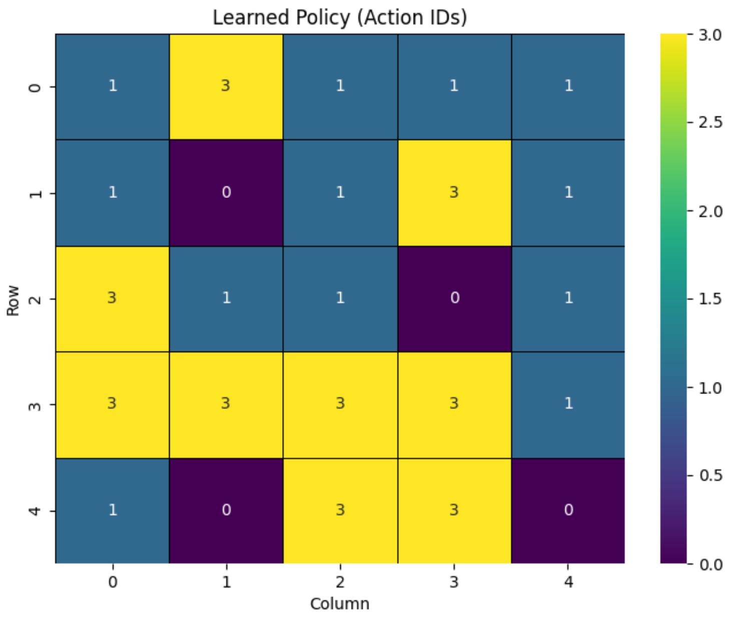

Fig. 4: Learned Policy (Action IDs)[3].

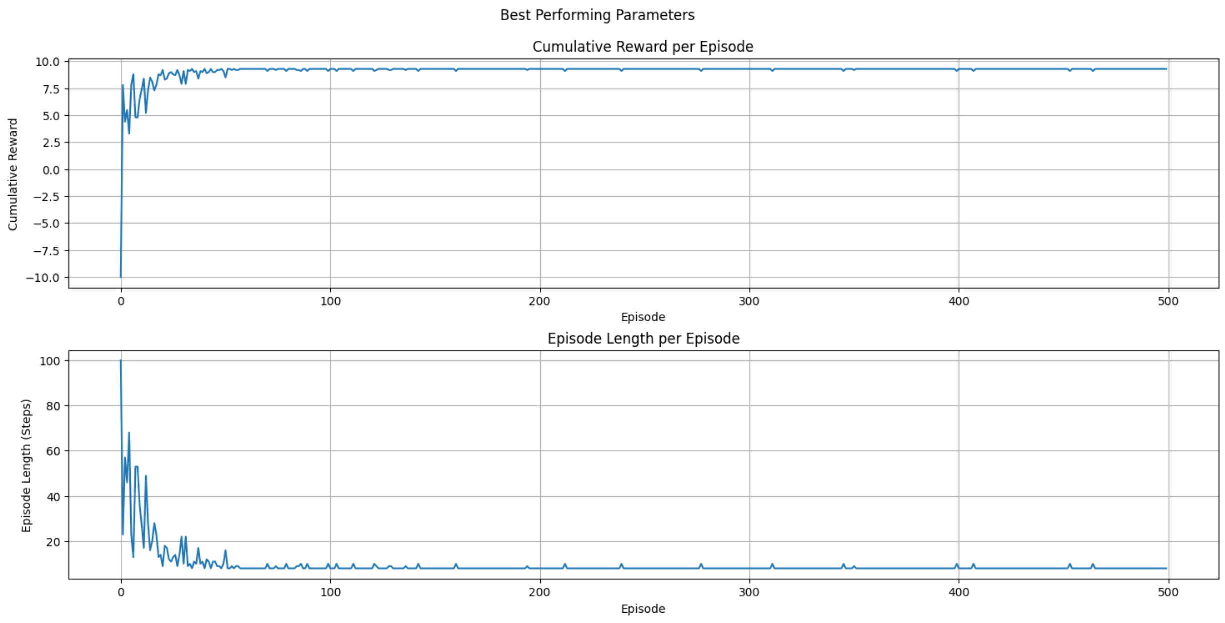

Fig. 5: Best performance parameters[3].

Looking at the Best performance parameter plot, the parameters clearly shows that the Q-learning reached convergence after around 100 episodes with close to around 9 cumulative reward. Therefore, the results of analyzing these plots helps to determine how well Q-learning solves the grid world. The episode length plot suggests that initially, it takes the agent around 100 episodes, but once the agent learns more, it is able to solve the world by 10 episodes. The setting can be further refined to have a parameter sweep to have wider range to obtain a more efficient and optimum policy.

Advanced

7 Enhanced DQN Implementation

CartPole-v1 Environment: As shown in figure 6 the Dueling DQN agent was trained and evaluated in the OpenAI Gym's CartPole-v1 environment [4]. In this environment, a pole is attached to a cart that can move horizontally along a frictionless track. The agent controls the cart by applying a force to the left or right, with the goal of keeping the pole balanced upright.

The state space is continuous, consisting of the cart's position and velocity, as well as the pole's angle and angular velocity. The reward is +1 for every time step the pole remains balanced. An episode ends when the pole falls beyond a certain angle, the cart moves too far from the center, or the maximum episode length is reached[5]. **Dueling DQN Architecture:**The core of this

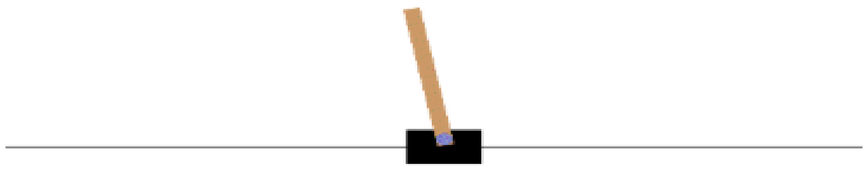

Fig. 6: **Cartpole** is a pole attached by an un-actuated joint to a cart[3].

implementation lies in the *Dueling DQN* architecture, a neural network designed to disentangle the estimation of state values and action advantages [6].
Unlike standard DQN, which directly approximates Q-values, Dueling DQN uses two separate streams: a value stream $V(s)$ that estimates the overall goodness of being in a state s, and an advantage stream $A(s,a)$ that estimates the relative benefit of taking action a in state s compared to other actions. These streams are then combined to estimate the Q-values $Q(s,a)$.
This decoupling often leads to faster and more stable learning because the agent can learn the value of states without necessarily having to learn the impact of each action, making the policy more sample-efficient (see Figure 7).
Experience Replay: To further stabilize training and break correlations in sequential data, experience replay is utilized [8]. During interactions with the environment, the agent stores transitions $(s, a, r, s', \text{done})$ in a replay buffer. When updating the Q-network, a mini-batch of experiences is randomly sampled from this buffer. This random sampling reduces the variance of updates and prevents the network from overfitting to the most recent experiences, leading to more robust and generalized learning. The replay buffer capacity (see Figure 8) determines the number of past experiences the agent remembers, influencing the stability and sample efficiency of training.

8 Analysis of Enhanced DQN Results

The 9 plot illustrates the agent's learning progress in the CartPole-v1 environment [5]. Early episodes show low and unstable rewards due to random exploration. Around episode 300, a clear upward trend emerges, reflecting a shift to exploiting learned strategies. Occasional dips in reward—even later in training suggest the agent encounters unfamiliar states or instabilities in its policy. The 50-episode moving average smooths these fluctuations and shows steady progress toward the maximum reward of 500, indicating near-optimal performance. However, not reaching 500 consistently hints at residual exploration or lack of full robustness. The 10 plot tracks how

well the agent estimates Q-values. High initial loss reflects early inaccuracies, which decrease as learning progresses. The plot isn't perfectly smooth; spikes indicate moments of instability, possibly from new state encounters or major policy updates. These fluctuations suggest areas for tuning or improving stability. The moving average highlights the overall decline in loss, confirming that the agent's predictions become more accurate over time.

9 RLlib Algorithm on Atari Environment

The agent interacts with the Atari Pong environment (Figure 11), receiving pixel data representing the game screen (i.e., the state) [10, 11]. The goal is to solve a high-dimensional, visually rich

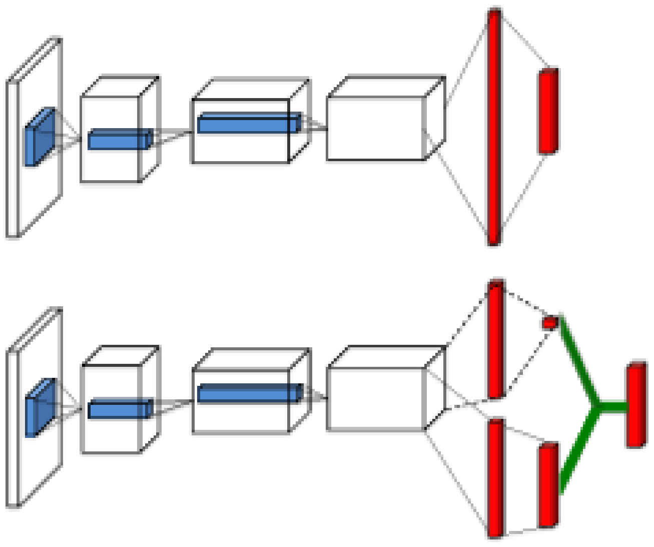

Fig. 7: A popular single stream Q-network (top) and the dueling Q-network (bottom)[7].

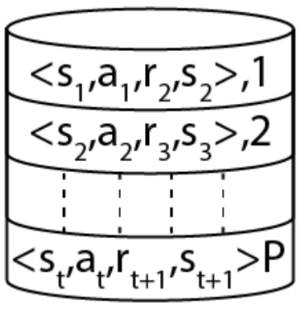

Fig. 8: **Experience Replay** is a replay memory technique where we store the agent's experiences at each time-step [9].

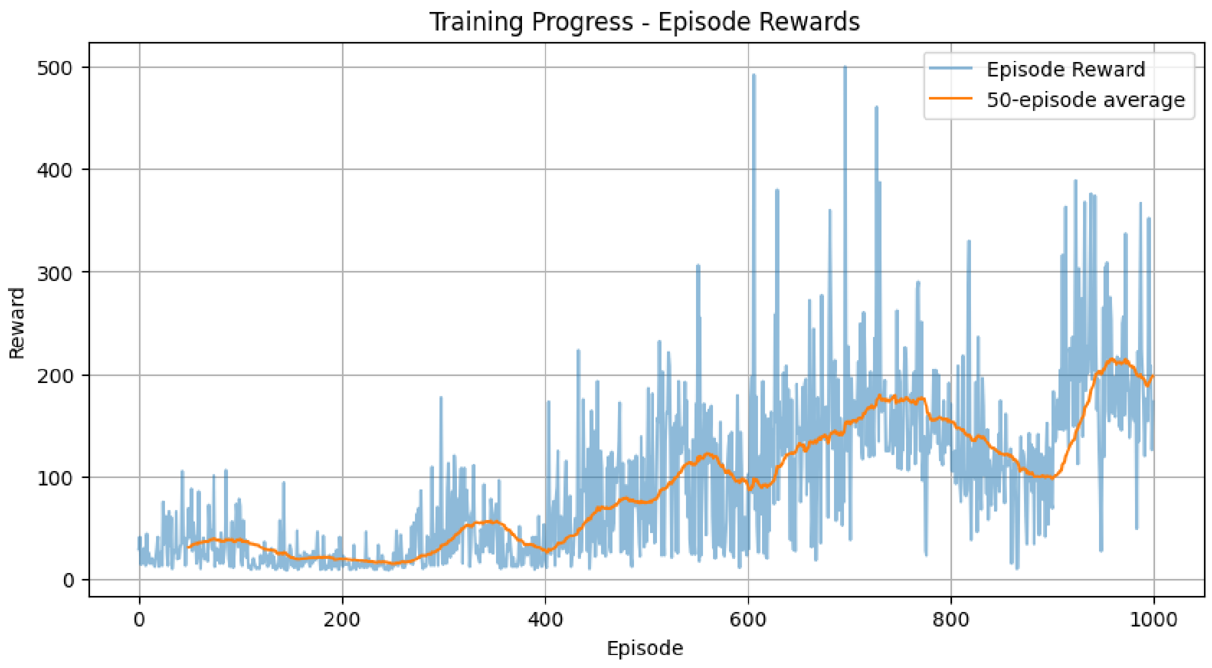

Fig. 9: Training Progress - Episode Rewards][3].

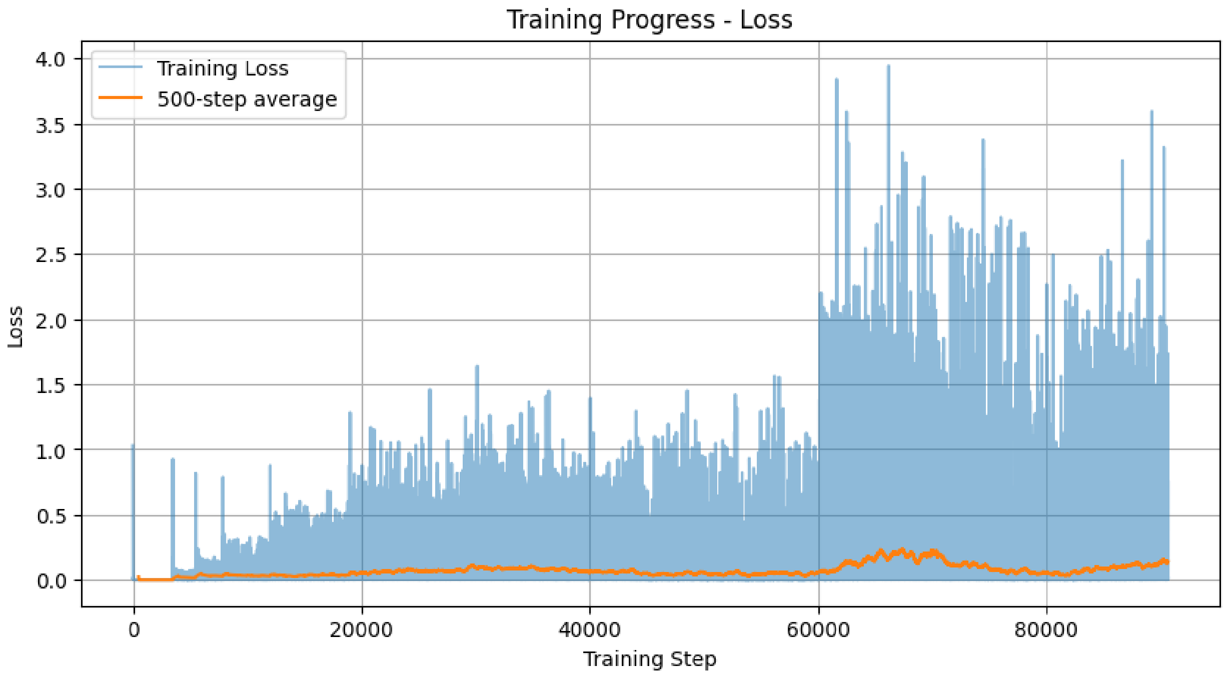

Fig. 10: Training Progress - Loss][3].

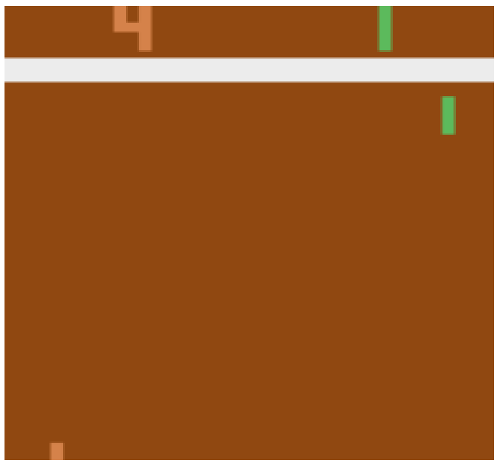

Fig. 11: Training progress for Atari Pong using PPO [3].

task where the agent controls a paddle to volley a ball, aiming to maximize its score.
Raw pixel input requires effective feature extraction to enable the agent to move the paddle and win points by preventing the ball from passing. This environment is high-dimensional and requires the agent to learn directly from visual input, presenting a significant challenge for convolutional neural networks (CNNs). The main difficulty lies in learning temporal dependencies (e.g., the ball's trajectory) and strategic paddle movements to outperform the opponent. **Proximal Policy**

Optimization (PPO) directly optimizes a neural network policy to maximize rewards in Atari Pong. It is a policy gradient method chosen for its stability and efficiency, updating the policy iteratively.
A key component of PPO is the *clipped surrogate objective function*, defined as:

$$L(\theta) = \mathbb{E}_t \left[\min \left(r_t(\theta) A_t, \, \text{clip}(r_t(\theta), 1 - \epsilon, 1 + \epsilon) A_t \right) \right] \tag{4}$$

Here, $r_t(\theta)$ is the probability ratio between the new and old policies, and A_t is the advantage function. The function `clip` ensures that $r_t(\theta)$ stays within a trust region defined by $\epsilon = 0.1$, limiting large policy updates and promoting stable learning.
The policy network maps game states (pixel frames) to actions (paddle movements). PPO's

clipped objective prevents divergence in high-dimensional environments like Atari by enforcing conservative updates.
Convolutional layers extract visual features from raw pixel inputs, enabling the fully connected layers to determine the optimal action and estimate expected rewards more effectively.

10 Analysis of RLlib Results

The mean episode return (Figure 12) starts around −20 and quickly rises to about 20, plateauing near the 20,000-step mark and remaining stable afterward. This indicates that the agent learned Pong quickly, improving rapidly and then converging to a strong, though possibly not optimal, policy. A return near 20 suggests the agent consistently wins by a large margin, reliably outperforming the AI opponent.

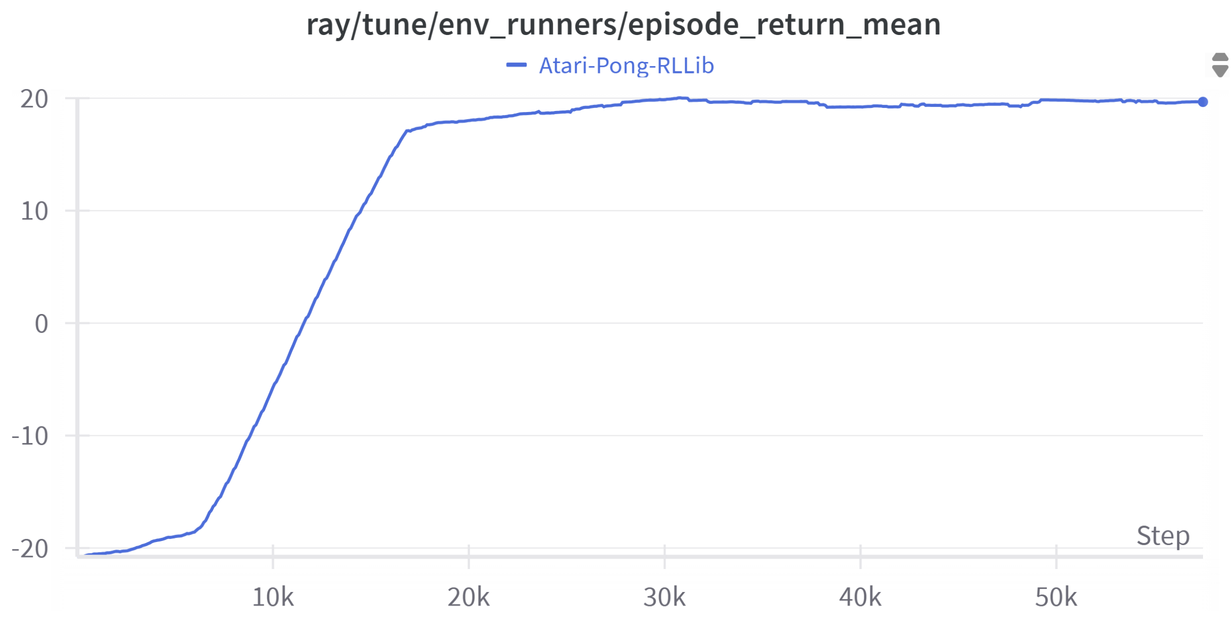

Fig. 12: Mean episode return over time [3].

The mean episode length (Figure 13) starts around 800 steps and quickly rises to about 2200, plateauing near the 20,000-step mark. This mirrors the return trend and implies that the agent learned to sustain longer rallies. The simultaneous plateau in both return and length suggests consistent, effective gameplay, reinforcing that the agent has developed a strong policy. The

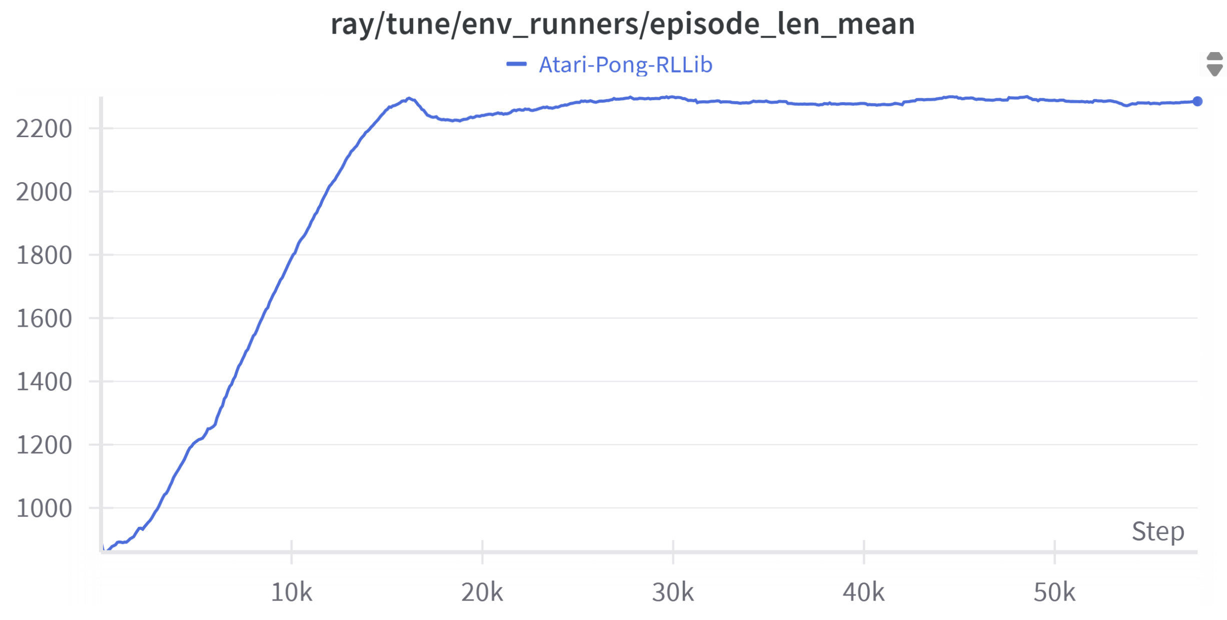

Fig. 13: Mean episode length over training steps [3].

value function explained variance (Figure 14) starts near zero and rises quickly to the range of 0.8–0.9 by 10,000 steps, then remains stable. This suggests that the value function becomes increasingly accurate at predicting future rewards, improving learning stability and performance. The value function loss (Figure 15) starts high (approximately 0.08) and quickly drops to near

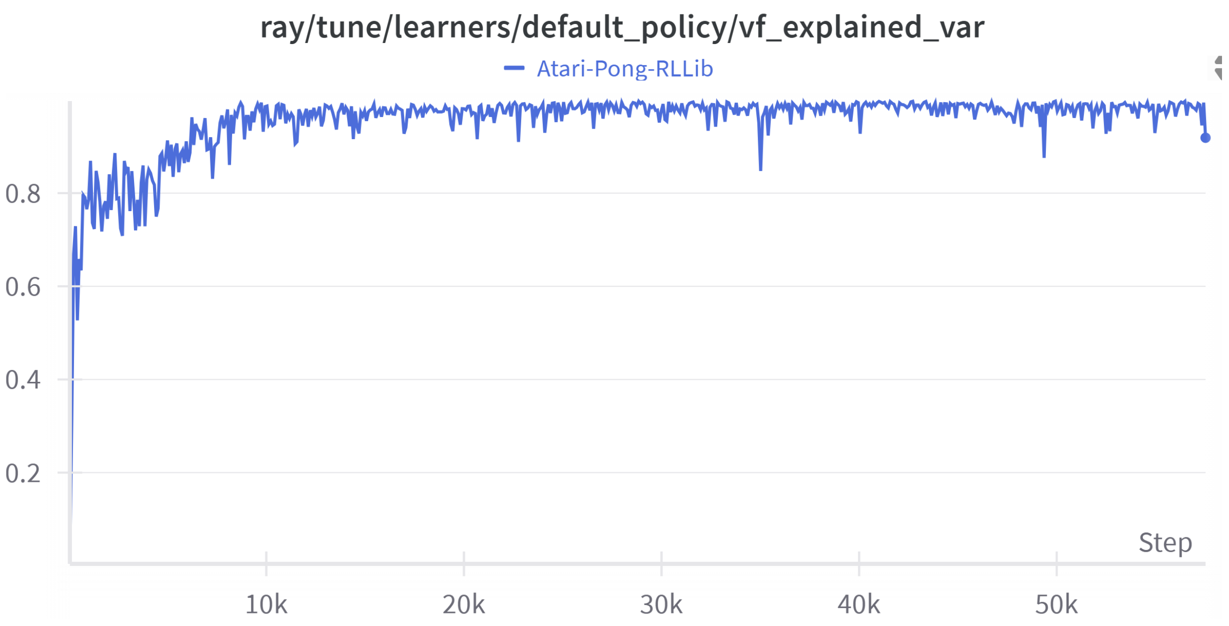

Fig. 14: Explained variance of the value function during training [3].

zero by 10,000 steps, remaining low afterward. This indicates the agent is increasingly accurate at evaluating states, in alignment with the explained variance results. The policy loss (Figure 16)

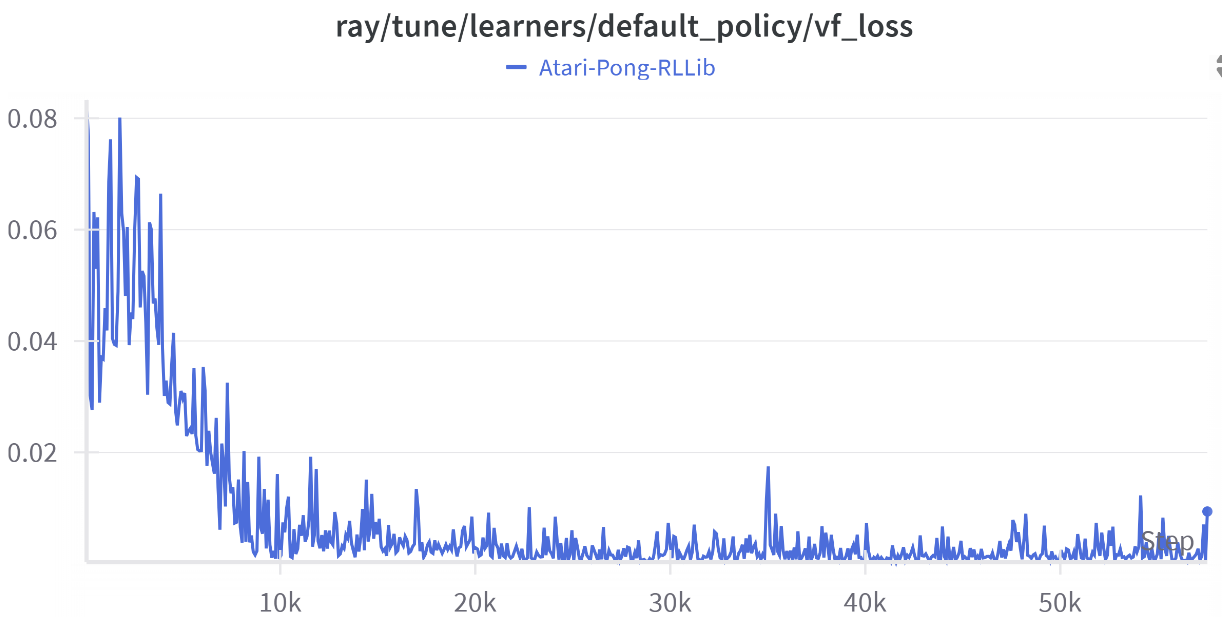

Fig. 15: Value function loss during training [3].

fluctuates throughout training without a clear downward trend, indicating ongoing exploration and policy refinement. These oscillations suggest that the agent is near an optimal policy, making further improvements more incremental. Kullback-Leibler (KL) divergence, or KL loss, measures the divergence between two probability distributions. In reinforcement learning, it is often used to quantify the difference between the current and previous policies. A low KL loss indicates small, stable policy updates, while high values can signal instability or significant exploration.
In this project, the mean KL loss (Figure 17) starts at zero, gradually increases to 0.02–0.04, and occasionally spikes above 0.1. This indicates mostly small updates with occasional bursts of

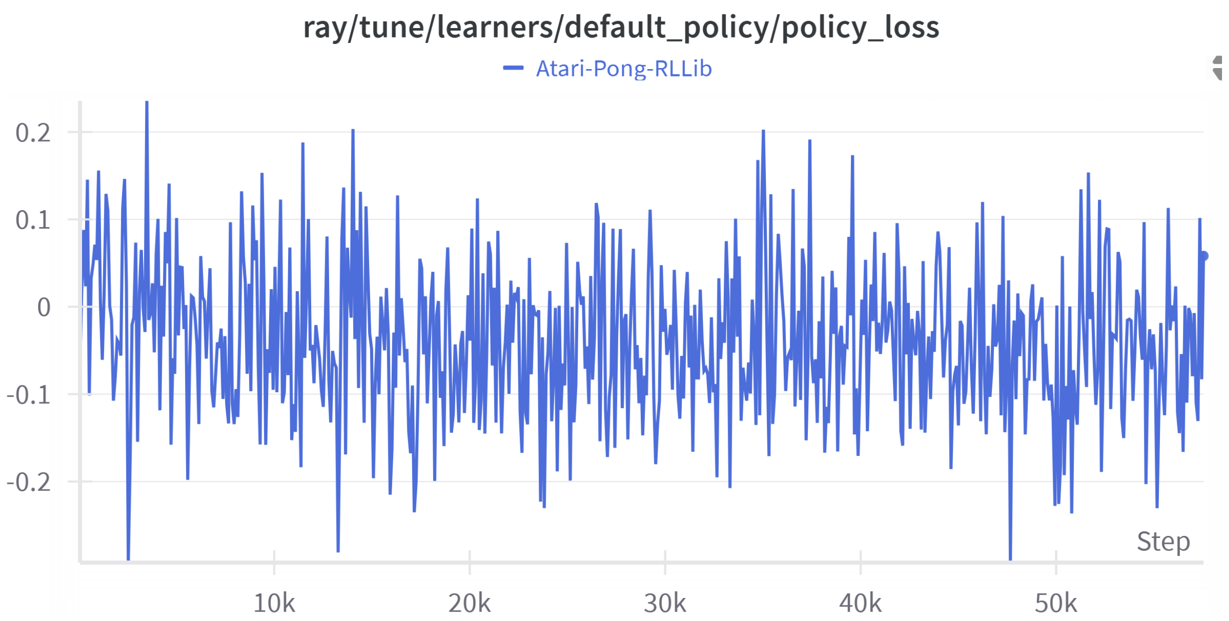

Fig. 16: Policy loss evolution [3].

exploration. The stable values suggest a balanced trade-off between exploration and exploitation, although limited policy updates might hinder further progress.

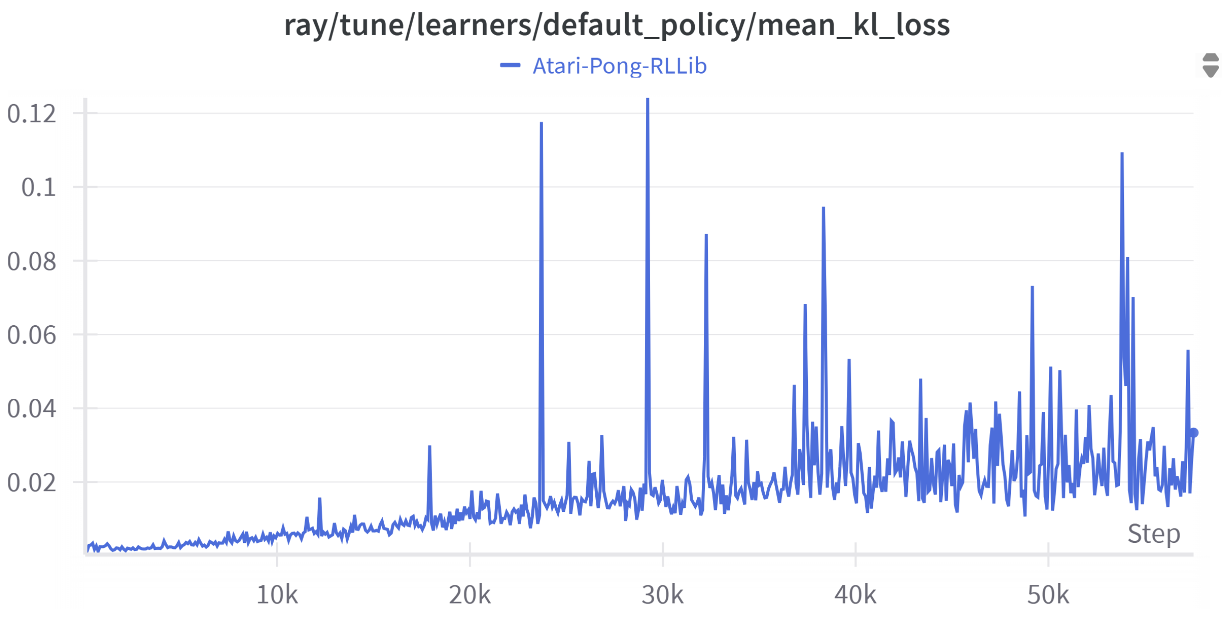

Fig. 17: Mean KL divergence during training [3].

RAM utilization begins around 40% and gradually stabilizes in the 41–42% range, suggesting a low and consistent memory footprint. CPU utilization stays roughly constant around 70%, indicating that training is CPU-bound but stable.

Extra Credit - Additional Implementations

11 Implementation of PPO or SAC

The Soft Actor-Critic (SAC) algorithm, employed in this study, is an off-policy, actor-critic reinforcement learning method specifically designed to maximize both the expected cumulative reward and the entropy of the learned policy [12]. This is achieved by optimizing a stochastic policy, $\pi(a|s)$, to maximize the following objective function:

$$J(\pi) = \mathbb{E}\left[\sum_{t=0}^{T} \gamma^t \left(R(s_t, a_t) + \alpha \cdot \mathcal{H}(\pi(\cdot|s_t))\right)\right] \tag{5}$$

The expectation in Equation 5 is over trajectories $\tau \sim \pi$.
Where:

- ▷ τ represents a trajectory sampled from the policy π.
- ▷ γ is the discount factor, weighting future rewards.
- ▷ $R(s_t, a_t)$ is the reward received at state s_t upon taking action a_t.
- ▷ α is the temperature parameter, which governs the relative importance of the entropy term $\mathcal{H}(\pi(\cdot|s_t))$. Higher α values encourage more exploration by favoring higher-entropy policies.
- ▷ $\mathcal{H}(\pi(\cdot|s_t))$ denotes the entropy of the policy at state s_t, measuring the randomness or uncertainty of the action distribution.

SAC utilizes two Q-functions, $Q_1(s, a)$ and $Q_2(s, a)$, to mitigate overestimation bias common in Q-learning approaches. These Q-functions are iteratively improved using a bootstrapped estimate based on the Bellman equation:

$$Q(s, a) = r + \gamma \cdot \mathbb{E}_{s' \sim P}\left[V(s')\right] \tag{6}$$

Where:

- ▷ r is the immediate reward.
- ▷ P is the transition dynamics of the environment.
- ▷ $V(s')$ is the value function, which estimates the expected return from state s'.

The value function is defined as:

$$V(s') = \mathbb{E}_{a' \sim \pi}\left[Q(s', a') - \alpha \log \pi(a'|s')\right] \tag{7}$$

This formulation in Equation 7 incorporates the entropy bonus, encouraging the agent to explore diverse actions.
Finally, the policy is updated to minimize the following loss function:

$$J_\pi(\theta) = \mathbb{E}_{s \sim D, a \sim \pi}\left[\alpha \log \pi(a|s) - Q(s, a)\right] \tag{8}$$

Where:

- ▷ D represents the replay buffer, a memory store of past experiences used for off-policy learning.
- ▷ θ represents the policy network parameters.

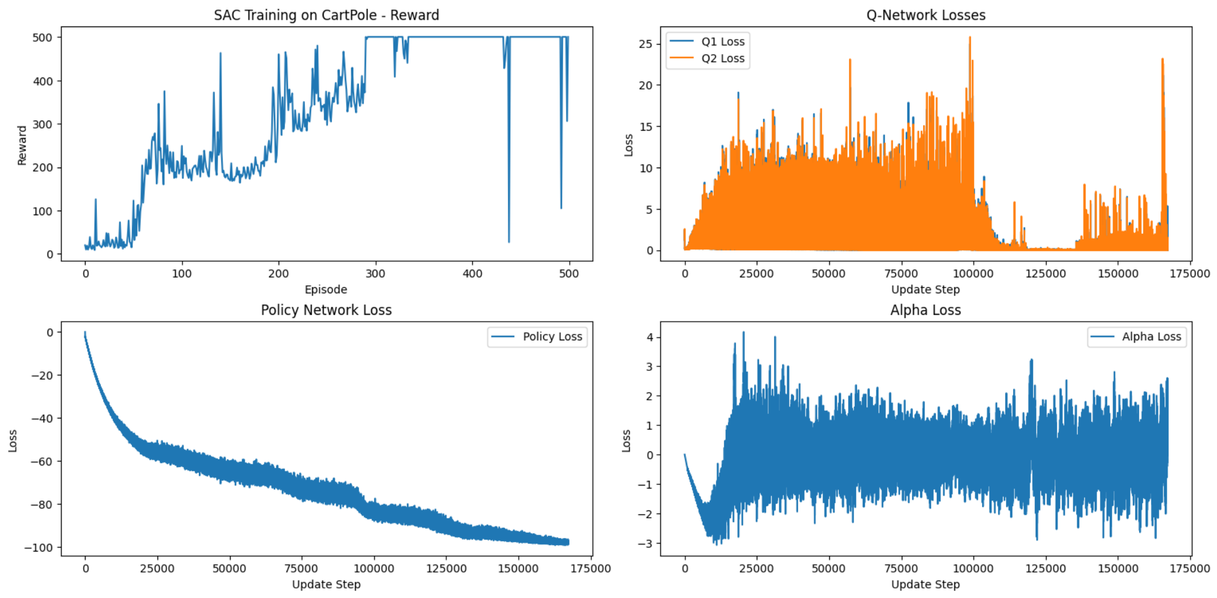

Fig. 18: SAC Training on CartPole - Reward and Losses[3].

This loss function in Equation 8 encourages the policy to choose actions with high Q-values while simultaneously maintaining high entropy, balancing exploitation and exploration. The automatic tuning of the α parameter maintains an appropriate level of entropy during training.
Examination of the training process and generated plots, Figure 18, provides evidence of successful learning by the SAC agent within the `CartPole-v1` environment. The episode reward plot demonstrates a clear trend of increasing rewards over training episodes, converging towards the maximum possible reward, which indicates the agent is learning an optimal policy. The reward curve appears jumpy at times, suggesting that the agent is still exploring or adapting its policy to uncertain states. The Q-network loss plots (Figure 18) for Q_1 and Q_2 show a decline in loss values over time, suggesting that the Q-functions are effectively learning to approximate the optimal Q-values for state-action pairs. While both loss curves generally decrease, occasional fluctuations are visible, reflecting the inherent stochasticity of the environment and the ongoing refinement of Q-value estimates.

The policy loss plot (Figure 19) similarly demonstrates a downward trend, indicating that the policy network is increasingly selecting actions that yield high expected return while maintaining adequate entropy to support exploration.
However, it is important to note that the implementation applies a discretional technique to convert continuous SAC outputs into discrete actions. This introduces a limitation — the agent may not be able to perform actions that lie between the predefined discrete options. As a result, optimal performance could be constrained in environments where nuanced or intermediate actions are critical.

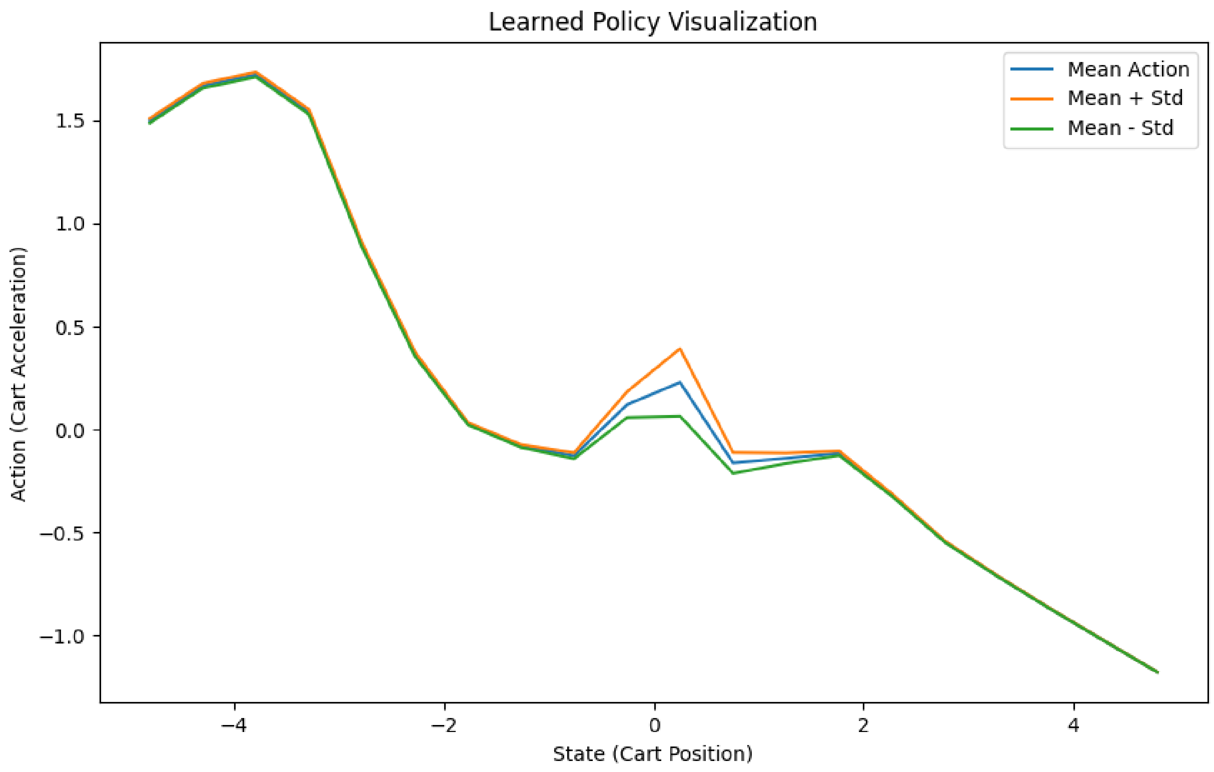

Fig. 19: Learned Policy Visualization[3].

References

[1] R. S. Sutton, A. G. Barto, *et al.*, *Reinforcement learning: An introduction*, vol. 1. MIT press Cambridge, 1998.

[2] C. J. Watkins and P. Dayan, “Q-learning,” *Machine learning*, vol. 8, pp. 279–292, 1992.

[3] A. Mostafavi, “https://github.com/Anndischeh/Deep-Reinforcement-Learning-Coursework,” 2025.

[4] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, “Playing atari with deep reinforcement learning,” *arXiv preprint arXiv:1312.5602*, 2013.

[5] G. Library, “Cartpole-v1 environment documentation.” `https://www.gymlibrary.dev/environments/classic_control/cart_pole/`, 2024. Accessed: 2025-05-09.

[6] Z. Wang, T. Schaul, M. Hessel, H. Hasselt, M. Lanctot, and N. Freitas, “Dueling network architectures for deep reinforcement learning,” in *International conference on machine learning*, pp. 1995–2003, PMLR, 2016.

[7] A. Mohamed Ahmed, T. T. Nguyen, M. Abdelrazek, and S. Aryal, “Reinforcement learning-based autonomous attacker to uncover computer network vulnerabilities,” *Neural Computing and Applications*, vol. 36, no. 23, pp. 14341–14360, 2024.

[8] L.-J. Lin, *Reinforcement learning for robots using neural networks.* Carnegie Mellon University, 1992.

[9] S. Ravichandiran, *Hands-on reinforcement learning with Python: master reinforcement and deep reinforcement learning using OpenAI gym and tensorFlow.* Packt Publishing Ltd, 2018.

[10] Ray Team, “Rllib: Scalable reinforcement learning.” `https://docs.ray.io/en/latest/rllib/index.html`, 2024. Accessed: 2025-05-09.

[11] G. Library, “Pong-v5 environment documentation.” `https://www.gymlibrary.dev/environments/atari/pong/`, 2024. Accessed: 2025-05-09.

[12] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, “Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor,” in *International conference on machine learning*, pp. 1861–1870, Pmlr, 2018.