Atari Pong Reinforcement Learning

Final Project Report
ECS 170 FQ 2025

Vicente Aguayo, Ella Bourassa, Shantanu Deshpande, Palina Karzhenka, Grace Lim, Paul Ling, Lingfeng Ren, Heqi Zhao, Sam Yin

December 10, 2025

1 Introduction

The objective of the project was to build three different reinforcement learning (RL) agents for playing the Atari Pong game. The major goals were to learn about reinforcement learning and evaluate the performance of different RL algorithms in learning an effective game-playing policy. Through a comparative analysis, we aim to identify the relative strengths and trade-offs of each approach in terms of final performance and learning stability.

2 Background

Reinforcement Learning is a machine learning (ML) paradigm where an agent learns by interacting with an environment. This is different from the supervised learning ML paradigm, which relies on data with predefined examples of good behavior. In RL, an agent selects an action based on its current state, then receives a reward which may be a positive or negative feedback, and finally transitions to a new state. The goal of the agent is to learn an effective policy, an optimal "map" of states to actions, that maximizes expected cumulative discounted reward. This trial and error approach is useful for tasks where it may be difficult to collect examples of good behavior.

2.1 RL for Game AI

Games are an interesting benchmark for testing RL algorithms because they provide a well-defined but complex environment with clear rules and objectives. The Arcade Learning Environment (ALE)[1], which includes the classic Atari Pong, offers this standardized platform for evaluating agents using only raw pixel inputs. In this environment, success would require an agent to learn the visual features and plan actions over extended time, all of which tests an algorithm's generalization ability. As mentioned earlier, our project focuses on Atari Pong which is a two-dimensional tennis-like game. Although this may be considered a "solved" problem, we found it to be a great introductory task due to its simple, discrete action space and clear reward structure. Learning an effective policy in Atari Pong would require the agent to learn ball physics, anticipate the adversary's movement, and develop a strategic movement plan.

[1] "ALE Documentation," Farama.org, 2023. https://ale.farama.org/environments/pong/

3 Methodology

In our comparative analysis, we will be focusing on three RL algorithms with their own characteristics and trade-offs. We implemented model-free algorithms that require agents to actively learn, meaning they take actions and learn the utility function directly from experience.

3.1 Deep Q-Network (DQN) Agent

Our DQN agent implements an off-policy, value-based RL algorithm that combines Q-Learning with deep neural network function approximation to learn optimal policy directly from high-dimensional visual inputs.

Our implementation follows a standard DQN architecture with three key features:

a. Experience replay: the agent stores transitions in a replay buffer to randomly sample from it during training and prevent catastrophic forgetting.
b. Separate target network: a target Q-network is updated periodically to provide stable learning targets and prevent divergence
c. Frame stacking: the input is made up of four consecutive and stacked RGB frames, which allows the agent to perceive motion and velocity.

The DQN agent interacted with the following environment specifications:

a. Environment: Atari Pong (ALE/Pong-v5) through the Arcade Learning Environment interface.
b. Observation Space: Raw RGB pixel frames that are converted to 84x84 grayscale frames by Stable-Baselines3's internal preprocessing[2].
c. Action Space: Discrete action space with 6 possible actions based on the full Atari action set supported by the Arcade Learning Environment.
d. Reward Signal: Sparse and delayed rewards, such as +1 for scoring, -1 for conceding, and 0 otherwise.
e. Training Data: The agent generated its own training data through environment interaction, requiring 2 million timesteps.

This work relied on the following third-party dependencies:

a. Gymnasium/ALE: Provided the standardized Atari Pong environment interface and game emulation.
b. Stable-Baselines3: Supplied the core DQN implementation, training utilities, and environment wrappers.
c. NumPy and Matplotlib: Used for numerical computations and visualization of training metrics.

We also wrote several functions that define and aid the training pipeline:

1. `train_dqn_pong()`
    a. Handles environment creation with parallel processing.
    b. Manages directory structure for models and logs.
    c. Coordinates the training loop with progress tracking.

[2] "DQN — Stable Baselines3 1.5.1a9 documentation," stable-baselines3.readthedocs.io. https://stable-baselines3.readthedocs.io/en/master/modules/dqn.html

2. `plot_training_results()`
    a. Loads trained models using Stable-Baselines3's serialization.
    b. Plots the tracked reward vs. timesteps.
3. `load_and_test_model()`
    a. Loads trained models using Stable-Baselines3's serialization.
    b. Renders the trained agent and provides visualization of agent behavior.
4. `resume_training()`
    a. Loads the trained model and continues training for additional 50,000 timesteps.
    b. Utilized before we were able to train in 8 parallel environments.

Furthermore, here are the parameters of our DQN agent:

a. Neural network architecture: The agent uses a convolutional neural network policy automatically configured by Stable-Baselines3.
b. Hyperparameters:
    i. Learning rate: $2x10^{-4}$
    ii. Replay buffer size: 1,000,000 transitions
    iii. Batch size: 512
    iv. Discount factor: 0.99
    v. Exploration: Linear epsilon-greedy decay from 1.00 to 0.05 over 50% of training
    vi. Target network update: every 10,000 steps
    vii. Frame stacking: 4 frames
c. Training Configuration:
    i. Total timesteps: 2,000,000
    ii. Parallel environments: 8
    iii. Training frequency: every 4 steps with 4 gradient updates per rollout
    iv. Hardware acceleration: automatic GPU detection

3.2 State-Action-Reward-State-Action (SARSA) Agent

Our SARSA agent implements an on-policy, value-based reinforcement learning algorithm that updates its Q-values using the next action chosen under the current ε-greedy policy. This makes SARSA more conservative than DQN or Q-Learning and better aligned with environments where exploratory actions can destabilize performance, such as Atari Pong.

Our implementation follows a DQN-style architecture but replaces the target-network and replay-buffer components with an on-policy TD update. It includes three key features:

a. Frame preprocessing and stacking: `preprocess_obs()` converts raw RGB frames into 84×84 grayscale images, and the FrameStack class produces stacked 4-frame inputs that capture temporal information.
b. Deep convolutional Q-network: QNet provides a three-layer convolutional encoder followed by fully connected layers to estimate action-values over the Pong action set.
c. Flexible ε-greedy exploration: Exploration follows either linear or exponential decay (configured via Config), computed by the epsilon() method in SARSAgent.

The SARSA agent interacted with the following environment configuration:

a. Environment: Atari Pong (ALE/Pong-v5) via Gymnasium + ALE.
b. Observation Space: 210×160 RGB frames converted internally to stacked grayscale 84×84 frames.
c. Action Space: Discrete Atari Pong action set (up to 6 actions).
d. Reward Signal: +1 for scoring, −1 for conceding, and 0 otherwise.
e. Episode Length: Up to 5,000 steps.

We also implemented several functions that structure the training pipeline:

1. `train()`
    a. Manages ε-greedy exploration, TD updates, logging, evaluation, gradient clipping, and checkpointing.
2. `evaluate()`
    a. Runs greedy-policy evaluations over fixed episodes for performance monitoring.
3. `make_env()`
    a. Builds the Pong environment with frameskip, seeding, and action-space configuration.
4. `sarsa_update()`
    a. Performs the one-step SARSA TD update to train the Q-network.

The SARSA agent was trained with the following hyperparameters:

a. Learning rate: 1×10^{-4} with optional exponential decay
b. Discount factor: 0.99
c. Exploration: ε decays from 1.0 to 0.05 using linear or exponential schedules
d. Reward clipping: ±1
e. Frame stacking: 4 frames
f. Training length: 500 episodes
g. Evaluation frequency: every 25 episodes
h. Max episode length: 5,000 steps

3.3 Linear Function Approximation (LFA) + Gradient-Free Agent

Our LFA+Gradient-Free Agent implements a policy-search reinforcement learning algorithm that directly optimizes the parameters of a linear softmax policy using Cross-Entropy Method (CEM), instead of using temporal difference updates. By searching directly in parameter space and evaluating the entire policy on Atari Pong, this method removes the need of gradient, replay buffers, and target networks, while staying compatible with raw pixel observations.

Our implementation follows a simple linear architecture followed by a CEM loop. It includes three key features:

a. Feature reconstruction: The agent requires a 210*160 frame, downsamples them to 42*32, normalizes to [0,1], and computes a different image against the previous frame. The current frame, the difference image, and a constant bias term are concatenated into a single feature vector of size 42 * 32 * 2 * 1 = 2688.
b. Linear softmax policy: In each actions, using the dot product of feature vectors and the weight matrix `W ∈ ℝ^|A|x2688` to compute a preference, then pass the preferences through a softmax to obtain a stochastic policy from which the actual action is sampled and executed in ALE.
c. CEM-based parameter search: Instead of gradient update, the agent maintains a diagonal Gaussian search distribution over `W` with `W_std` and `W_mean`. At each generation, the top 20% "elite", evaluated in one or more Pong episodes, are used to update `W_std` and `W_mean`.

The LFA+Gradient-Free Agent interacted with the following environment configuration:

a. Environment: Atari Pong (ALE/Pong-v5) via Gymnasium + ALE.
b. Observation Space: a 2688 dimensional vector.
c. Action Space: Discrete Pong action set.
d. Reward Signal: +1 for scoring, −1 for conceding, and 0 otherwise.
e. Episode Length: Up to 20,000 steps.

We also implemented several cells that structure the training pipeline:

1. LFA
    a. Feature reconstruction, linear softmax policy setup.
2. Saves and reloads
    a. Saves `W_std` and `W_mean`, and `W_best` in `checkpoint/` directory and stores the training curve (`iter`, `elite_mean`, `best`, `std_mean`, `mean_steps`, etc.)
    b. Reloads the saved data to continue from where it was left off.
3. CEM
    a. Flexible number of executing generations of CEM training.

The LFA+Gradient-Free Agent was trained with the following hyperparameters:

a. Elite fraction: 0.2
b. Feature Dimension: 42*32
c. Reward Clipping: ±1
d. Training Length: 2500 CEM generations, with a total of 87 million cumulative steps
e. Evaluation Frequency: every episode
f. Max Episode Length: Up to 20,000 steps.

4 Results

4.1 DQN Agent

The DQN agent outperformed other models in our comparative analysis, specifically in time taken to converge and episode rewards. After 2 million timesteps, the mean episode reward was +21, meaning that the agent won every round of the game. The plot of episode rewards over timesteps can be seen below in Figure 1.

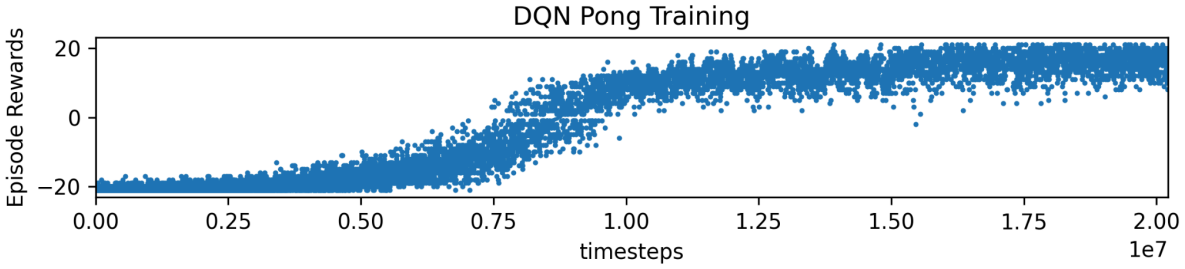


Figure 1: DQN Pong Training of Episode Rewards across total timesteps

The total timesteps to converge translates to a training time of around 2 hours, which was computed using a GPU rather than a CPU. It is worth noting that this method of parallel processing was specific to this agent's training and was not employed for the other agents. This could likely explain the difference in processing time for training across models.

4.2 SARSA Agent

The plot of episode rewards across total timesteps can be seen below in Figure 2.

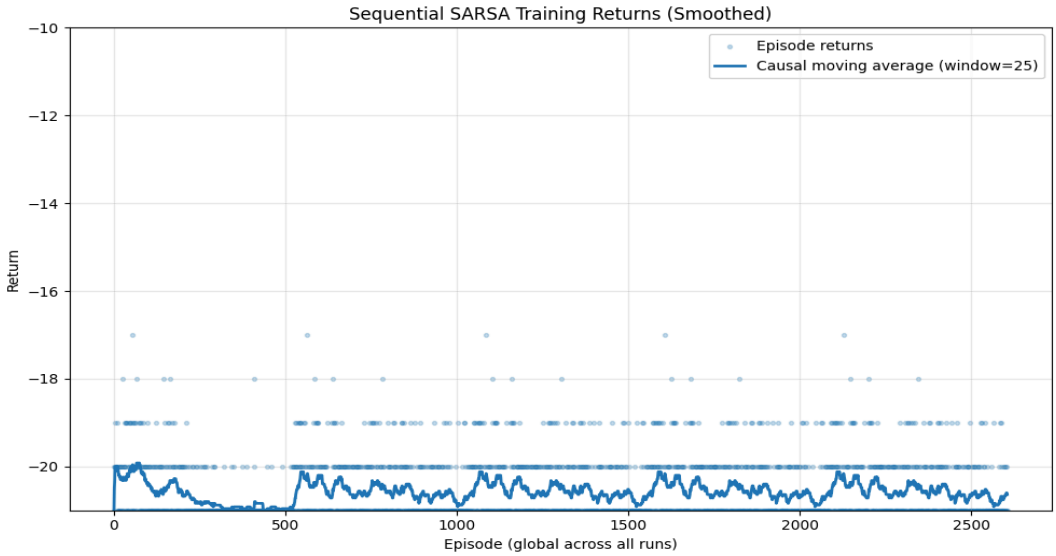
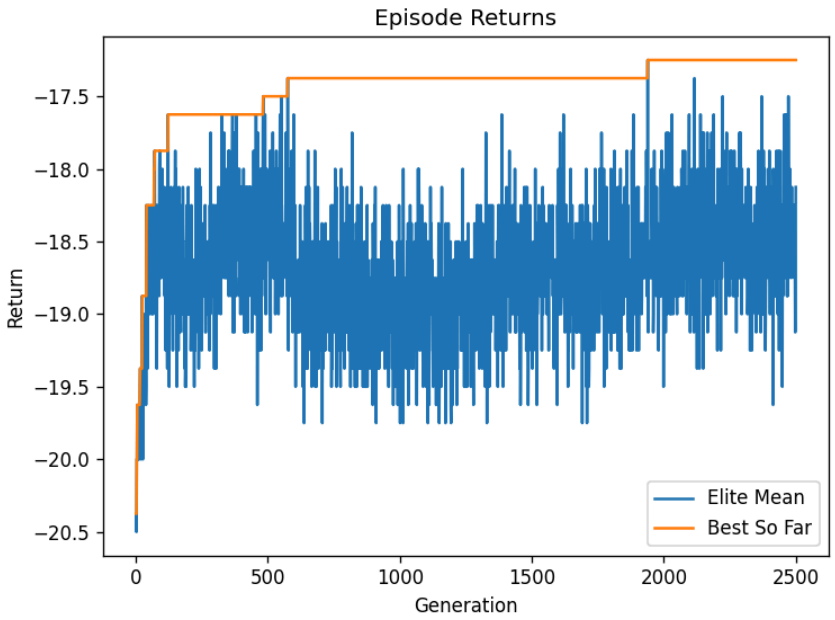


Figure 2: SARSA Pong Training of Episode Rewards across Total Timesteps

The SARSA agent showed the weakest performance among the models in our comparative analysis. Across 2,500 timesteps, significantly fewer than the other teams' training budgets, the

agent remained in the negative reward range and did not converge toward successful gameplay. Training took approximately two hours on a CPU, but the lack of experience replay, target network, and parallelization limited both learning speed and overall performance. In contrast, the other models, particularly DQN, achieved much higher episode rewards and demonstrated clear convergence, highlighting SARSA's relative inefficiency for high-dimensional control tasks like Atari Pong.

4.3 LFA + Gradient-Free Agent



The LFA+Gradient-Free agent, unlike the other two agents, searches directly in the parameter space. Over 2500 generations, we've improved from roughly -20.5 (random play) to -17.25, indicating that the CEM search is able to discover policies that perform noticeably better than random. However, the average lies around -18.75 with high variance. Training took approximately 10 hours on a CPU (~250min for 1000 gens). Unfortunately, this agent never reaches a positive score like the DQN agent achieved, even with tens of millions of environmental interactions. Compared with SARSA agent, the best case is slightly better but also has a higher episode length.

5 Discussion

Our results indicate that the DQN implementation outperformed both SARSA and LFA. DQN's agent's performance is important because as we can see, for a complex and pixel-based environment like Atari Pong, off-policy learning with experience replay offers an advantage in overcoming the challenges of learning the best policy from non-stationary targets.

A few shortcomings in our analysis are that agents were trained for different durations along with different processing power. This could potentially bias our results in making direct comparisons. However, comparing the algorithms' structures makes it clear that a more conservative algorithm

like SARSA, which doesn't implement experience replay, cannot be expected to perform better. Since that agent is likely to forget its initial experiences, learning will take longer, if it occurs at all.

6 Conclusion

In this project, we have successfully implemented three different reinforcement learning techniques as it applies to the Pong game. Overall, this project served as a comprehensive introduction to reinforcement learning. With the three different approaches, we were able to find practical limitations, like slow learning, in some algorithms but not others. Our comparative results exemplify how making an architectural design is just as important as the learning rule when trying to solve complex, high-dimensional environments.

7 Github

Here is the link for the GitHub titled "Atari Pong Reinforcement Learning" that includes the code for this project:

https://github.com/GL56/ECS170-RLProject

8 Contributions

8.1 Vicente Aguayo

Contributed significantly to LFA agent code, and writing of all major project reports and check ins.

8.2 Ella Bourassa

Contributed significantly to SARSA agent code, and writing of all major project reports and check ins.

8.3 Shantanu Deshpande

Contributed significantly to SARSA agent code, and writing of all major project reports and check ins.

8.4 Palina Karzhenka

Contributed significantly to DQN agent code, and writing of all major reports and check ins.

8.5 Grace Lim

Contributed significantly to DQN agent code, and writing of all major project reports and check ins.

8.6 Paul Ling

Contributed to write up of the project proposal on week 2.

8.7 Lingfeng Ren

Contributed significantly to DQN and policy gradient code, as well as, writing of all major project reports and check ins.

8.8 Heqi Zhao

Contributed significantly to LFA agent code, and writing of all major project reports and check ins.

8.9 Sam Yin

Contributed to write up of the project proposal on week 2.