Autonomous Agent for Atari Breakout using Deep Reinforcement Learning (SWE436)

By Mahir Al Shahriar 2019831077

October 27, 2025

1 Problem Statement

The core problem addressed in this project is to create an autonomous agent capable of learning to play the Atari game "Breakout" at a high level of performance using only raw pixel data.

1.1 Modelling the Problem as a Markov Decision Process (MDP)

To solve this, I modelled the game as a Markov Decision Process (MDP), which provides the standard framework for sequential decision-making. The key components of this MDP are:

- **State (s):** The current situation of the game. Using a single screen frame is inadequate because it lacks motion information. My solution defines the state as a stack of the four most recent preprocessed frames. This stacked state provides the agent with the necessary temporal context to infer the ball's direction and velocity.
- **Action (a):** The set of possible moves the agent can make. In the `ALE/Breakout-v5` environment, the agent has a discrete action space. My model, with `action_dim=4`, allows the agent to choose from the available game actions.
- **Transition Function (P):** The environment's internal dynamics that determine the next state s' given the current state s and action a.

This function is unknown and stochastic; the agent must learn the consequences of its actions purely through interaction.

- **Reward (R) :** The immediate feedback signal from the environment. In Breakout:
  - *Positive Reward (+1):* The agent receives a positive reward for breaking a brick.
  - *Negative Reward (-1) (Punishment):* The agent receives a negative reward when it fails to catch the ball and loses a life.
  - *Neutral Reward (0):* For all other time steps, the agent receives no reward.

The agent's objective within this MDP is to discover a policya strategy for selecting actions in each statethat maximizes the cumulative future discounted reward, known as the *episodic return*.

1.2 The Reinforcement Learning Loop

The learning process follows a fundamental cycle of interaction:

1. The agent observes the current state s_t (the stack of four frames).
2. Based on its current policy, the agent selects and executes an action a_t.
3. The environment transitions to a new state s_{t+1} and provides a reward r_t.
4. The agent uses this experience (s_t, a_t, r_t, s_{t+1}) to update its model, improving its decision-making policy.
5. This loop repeats continuously throughout the training process.

1.3 Contrast with Supervised Learning

This Reinforcement Learning approach is fundamentally different from Supervised Learning in several key aspects:

- **Objective:** The goal is to maximize *episodic return* over time, not to minimize a prediction error on a fixed dataset.

- **Data Generation:** The training data is not collected beforehand. Instead, it is generated through the agent's own actions, leading to a non-stationary and highly correlated stream of experiences.
- **Absence of a Static Dataset:** There is no pre-defined validation or test set. Performance is evaluated directly in the environment by measuring the total reward achieved per episode.
- **Interpretation of Loss:** The loss function (e.g., the Mean Squared Error loss of the DQN) is not a direct performance metric. Because the target Q-values are estimated by the same network being trained, the targets are non-stationary. A decreasing loss indicates the network's predictions are becoming more self-consistent, but it does not guarantee the agent is learning a better policy. Therefore, the true measure of success is the *episodic return* achieved during evaluation.

2 Methodology

I developed a Deep Q-Learning agent to solve the Atari Breakout environment using Python and TensorFlow/Keras, with training conducted on a Mac M1 CPU. The methodology builds upon Q-Learning principles enhanced with neural network function approximation and several stabilization techniques crucial for successful deep reinforcement learning.

2.1 Deep Q-Learning with Neural Network Approximation

The foundation of my approach is Q-Learning, a model-free off-policy algorithm that learns the optimal action-value function $Q^*(s, a)$, representing the maximum expected cumulative reward from state s after taking action a and following the optimal policy thereafter.

The standard Q-Learning update rule is:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right] \tag{1}$$

For high-dimensional state spaces like pixel-based observations, maintaining a Q-table is infeasible. I parameterized the Q-function using a deep neural network, where $Q(s, a; \theta)$ represents the network with weights θ that maps input states to Q-values for each action. This allows the agent to generalize across similar states and handle the complex visual input from the game environment.

2.2 Target Network and Moving Target Stabilization

A fundamental challenge in Deep Q-Learning is the "moving target" problem. When using the same network to both calculate the current Q-values and the target values $r_t + \gamma \max_a Q(s_{t+1}, a; \theta)$, the targets shift with every parameter update, creating unstable training dynamics.

I addressed this by implementing a **Target Network**, a separate network $Q(s, a; \theta^-)$ with identical architecture but frozen parameters. The target values are computed as:

$$\text{Target} = r_t + \gamma \max_a Q(s_{t+1}, a; \theta^-) \tag{2}$$

The target network's weights θ^- are periodically synchronized with the main Q-network's weights θ, typically every few thousand steps. This creates stable targets for multiple learning updates, dramatically improving training convergence.

2.3 Experience Replay and IID Sampling for Generalization

Reinforcement learning inherently generates sequential, temporally correlated experiences as the agent interacts with the environment. This correlation violates the Independent and Identically Distributed (IID) assumption fundamental to most stochastic optimization algorithms, leading to several problems:

1. **Catastrophic Forgetting:** The network tends to overfit to recent experiences and forget previously learned patterns.

2. **Inefficient Learning:** Correlated updates provide redundant information and poor gradient estimates.

3. **Poor Generalization:** The network fails to learn robust policies that work across diverse states.

Temporally

Figure 1: Correlated Observation Samples

I solved these issues by implementing an **Experience Replay Buffer**. This data structure stores transition tuples $(s_t, a_t, r_t, s_{t+1}, \text{done})$ as the agent interacts with the environment. During training, I sample random mini-batches from this buffer rather than using consecutive experiences.

This approach provides two crucial benefits for generalization:

- **Breaking Temporal Correlations:** By randomly sampling from diverse past experiences, the training data approximates IID conditions. This prevents the network from overfitting to recent trajectories and forces it to learn patterns that generalize across the entire state space.
- **Comprehensive State Coverage:** The replay buffer accumulates experiences from different phases of learning and various game situations. Sampling from this diverse dataset ensures the network encounters a wide distribution of states during training, leading to more robust policy learning and better generalization to unseen situations during evaluation.

The `ReplayBuffer` class with its `push` and `sample` methods forms the core of this mechanism, enabling the agent to learn from a diverse, decorrelated set of experiences.

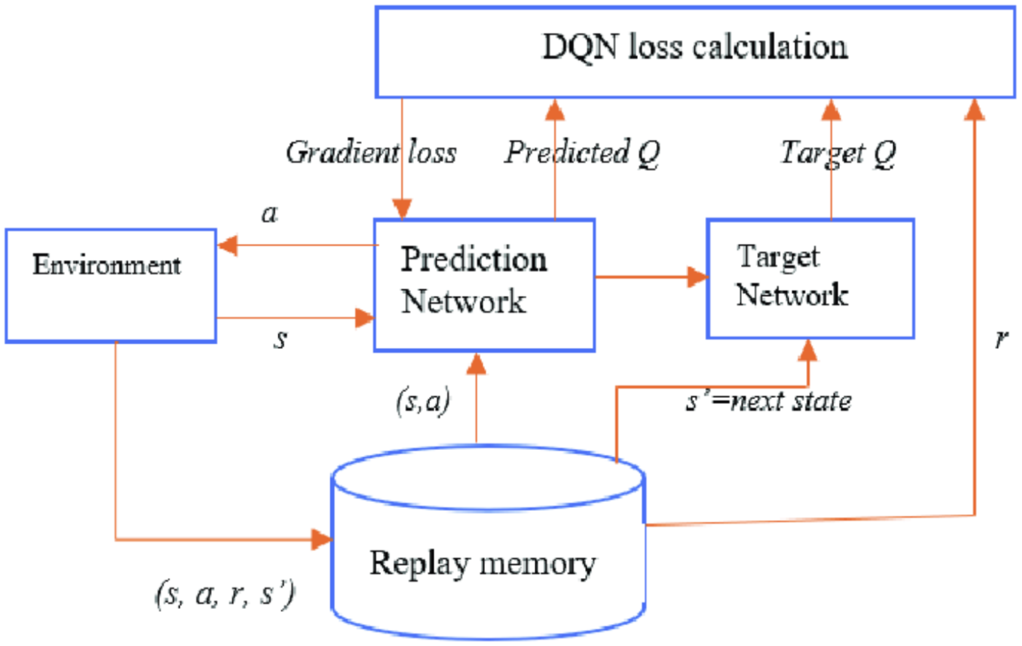

Figure 2: DFD for DQN with Experience Replay

2.4 Epsilon-Greedy Exploration Strategy

To balance exploration of new actions with exploitation of known rewarding strategies, I employed an **epsilon-greedy policy**. The agent selects random actions with probability ϵ and greedy actions (highest Q-value) with probability $1-\epsilon$. The exploration rate ϵ follows a linear decay schedule from an initial high value to a minimum value, encouraging extensive exploration early in training while progressively shifting toward exploitation as the policy improves.

This comprehensive methodology combines theoretical Q-Learning foundations with practical deep learning stabilization techniques to enable effective learning in complex environments.

3 Implementation Details

3.1 Data Preprocessing Pipeline

To transform the raw game frames into a format suitable for neural network processing, I implemented a comprehensive preprocessing pipeline that balances information preservation with computational efficiency.

3.1.1 Grayscale Conversion and Luminance Preservation

The original RGB frames (210 160 3) were converted to grayscale using the psychometrically weighted luminance formula:

$$Y = 0.299R + 0.587G + 0.114B \tag{3}$$

This specific weighting is crucial because it reflects human perceptual sensitivity - our visual system is most sensitive to green light (59% contribution), followed by red (30%), and least sensitive to blue (11%). This transformation serves multiple purposes:

- **Perceptual Relevance:** It preserves the most visually salient information while reducing computational complexity
- **Dimensionality Reduction:** The input channels decrease from three to one, significantly reducing the number of network parameters
- **Color Invariance:** It makes the agent invariant to color scheme changes that don't affect gameplay semantics

3.1.2 Frame Cropping and Resizing

I cropped the frames to remove non-essential screen regions, eliminating the top 34 pixels (containing score information) and bottom 16 pixels (containing borders), resulting in a clean 160 160 playing area. This focused the agent's attention on relevant game elements. The cropped frame was then resized to 84 84 pixels using nearest-neighbor interpolation, which preserves sharp edges and avoids introducing blurring artifacts that could obscure important game features.

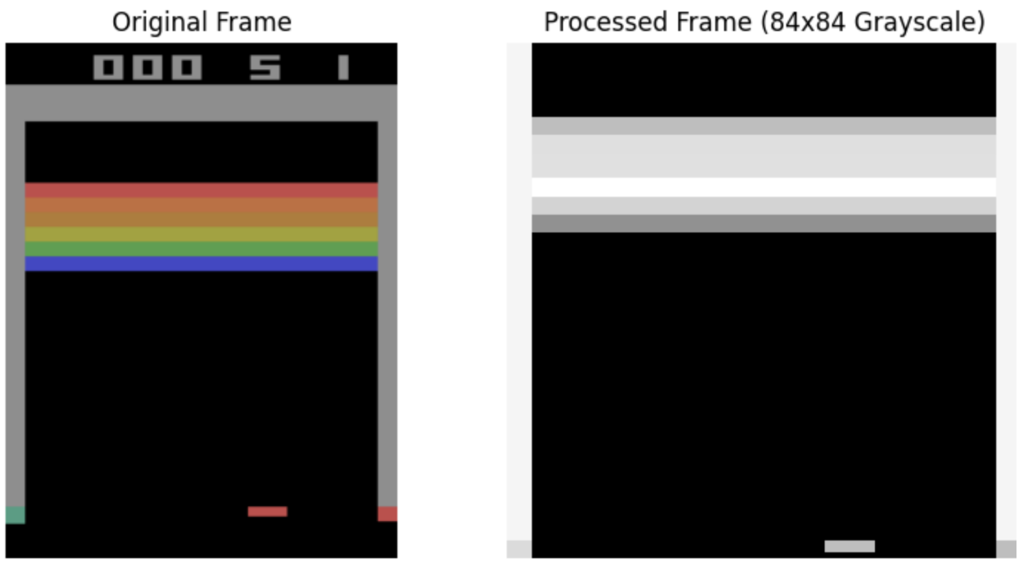

Figure 3: Original And Processed Frame

3.1.3 Temporal Information through Frame Stacking

A single frame provides only instantaneous spatial information, making it impossible to determine object dynamics like the ball's velocity and direction. To capture essential temporal dynamics, I implemented frame stacking, where the four most recent preprocessed frames are concatenated along the channel dimension. This creates a state representation of shape (84, 84, 4), providing the neural network with the necessary motion context to infer trajectories and make informed predictive decisions.

3.2 Neural Network Architecture and Design Choices

The Q-network architecture was carefully designed to extract hierarchical spatial features from the preprocessed frames while maintaining computational efficiency:

3.2.1 Convolutional Feature Extraction Layers

- **First Convolutional Layer:** 32 filters of size 88 with stride 4 and ReLU activation. This layer processes the input with large receptive fields to capture high-level spatial patterns like game object presence and rough positioning. The ReLU activation introduces non-linearity while maintaining computational efficiency and mitigating the vanishing gradient problem.

- **Second Convolutional Layer:** 64 filters of size 44 with stride 2 and ReLU activation. This layer extracts mid-level features from the previous layer's output, identifying more specific game elements and their spatial relationships.

- **Third Convolutional Layer:** 64 filters of size 33 with stride 1 and ReLU activation. This layer refines feature representations with smaller receptive fields, capturing fine-grained patterns and local interactions between game objects.

This hierarchical convolutional design enables the network to progressively learn from low-level visual features (edges, corners, basic shapes) to high-level game concepts (paddle positioning strategies, ball trajectory patterns, brick configuration analysis).

3.2.2 Decision-Making Layers

- **Flatten Layer:** Converts the 3D feature maps from the final convolutional layer (7764) into a 1D feature vector of 3136 elements, preparing the spatial features for dense processing.

- **Fully Connected Layer:** 512-unit dense layer with ReLU activation integrates the extracted spatial features into a comprehensive representation for value estimation. The ReLU activation maintains the network's ability to learn complex non-linear value functions.

- **Output Layer:** Linear layer (no activation) with 4 units representing the estimated Q-values for each possible action. Linear activation is used because Q-values can span arbitrary ranges and we need unbounded outputs for proper value estimation.

All layers use He normal initialization, which is particularly suitable for ReLU activations as it maintains stable variance through the network during forward and backward passes.

3.2.3 Loss Function Selection

I employed Mean Squared Error (MSE) as the loss function for Q-value regression.MSE provides stronger gradients for large errors, which can be beneficial during early training when the Q-value estimates are highly inaccurate. The MSE loss is defined as:

$$L = \frac{1}{N} \sum_{i=1}^{N} (Q_{\text{target}} - Q_{\text{predicted}})^2 \tag{4}$$

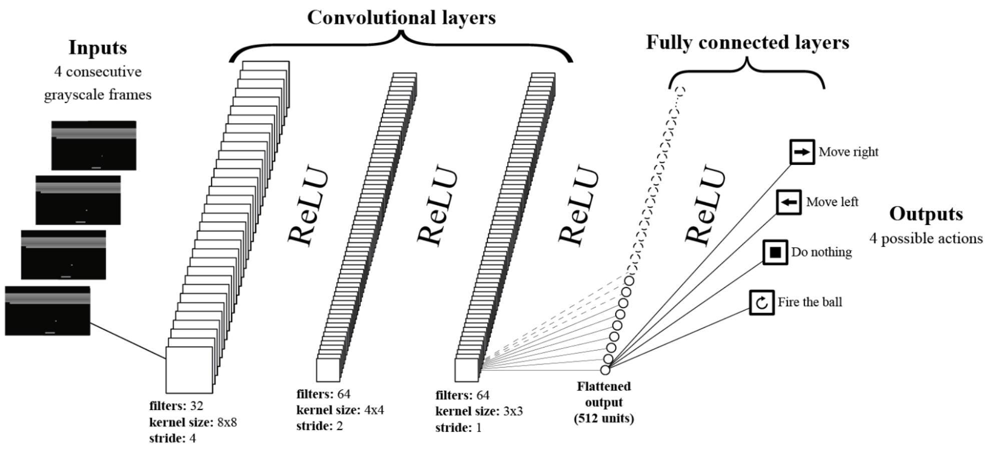

Figure 4: Deep Q Network

3.3 TensorFlow Data Flow and Computational Graph

The data undergoes a sophisticated transformation pipeline throughout the training process:

3.3.1 Preprocessing to Network Input

The transformation chain is: Raw RGB frames (210 160 3) Grayscale conversion (210 160 1) Strategic cropping (160 160 1) Resolution reduction (84 84 1) Temporal stacking (84 84 4) Value normalization (floating-point [0,1] range).

3.3.2 Network Computational Graph

The stacked, normalized state tensor flows through a carefully constructed computational graph:

- **Spatial Feature Extraction:** Each convolutional layer applies learned filters with specified strides, progressively reducing spatial dimensions while increasing feature depth and abstraction level
- **Feature Integration:** The flattened spatial features pass through dense layers that learn to combine these features for accurate value estimation
- **Gradient Computation:** TensorFlow's automatic differentiation and gradient tape mechanism efficiently compute gradients for the entire computational graph during backward passes

The `@tf.function` decorator on the `_train_step_tf` method compiles the training step into a static computational graph, providing significant performance improvements through operation fusion and optimized memory allocation.

4 Reinforcement Learning System Architecture

The training system implements a sophisticated data flow architecture that coordinates multiple concurrent processes to enable stable and efficient deep reinforcement learning. The architecture consists of four interconnected subsystems that operate in harmony throughout the training process.

4.1 Environment Interaction Subsystem

This subsystem handles the real-time interaction between the agent and the game environment, transforming raw sensory input into actionable state representations:

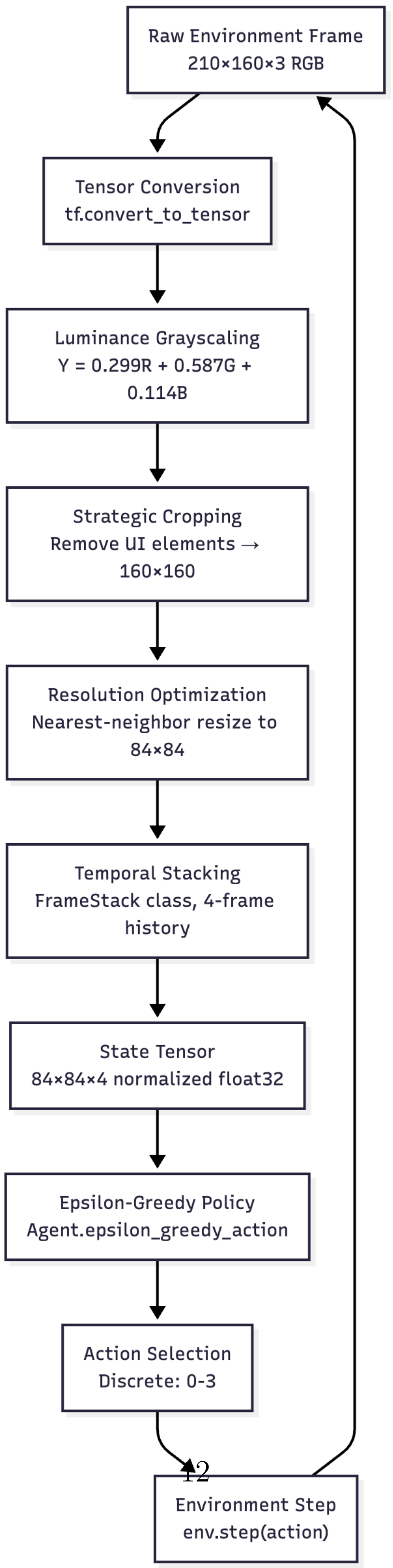

Figure 5: Environment Interaction Process

The `FrameStack` class maintains a deque of the four most recent frames, ensuring the agent always receives temporal context for motion perception. The epsilon-greedy policy dynamically balances exploration and exploitation, with exploration probability decaying linearly from 1.0 to 0.1 over 100,000 environment steps.

4.2 Experience Management Subsystem

This subsystem serves as the agent's long-term memory, storing and retrieving experiences to break temporal correlations and enable stable learning:

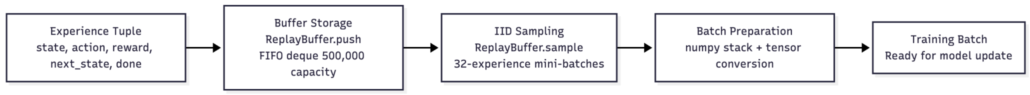

Figure 6: Experience Replay Process

The `ReplayBuffer` implements a fixed-size circular buffer that automatically evicts oldest experiences when capacity is reached. Random sampling from this diverse experience pool ensures the training data approximates independent and identically distributed conditions, crucial for preventing catastrophic forgetting and enabling robust generalization.

4.3 Learning and Optimization Subsystem

This core learning engine performs the neural network updates that drive policy improvement:

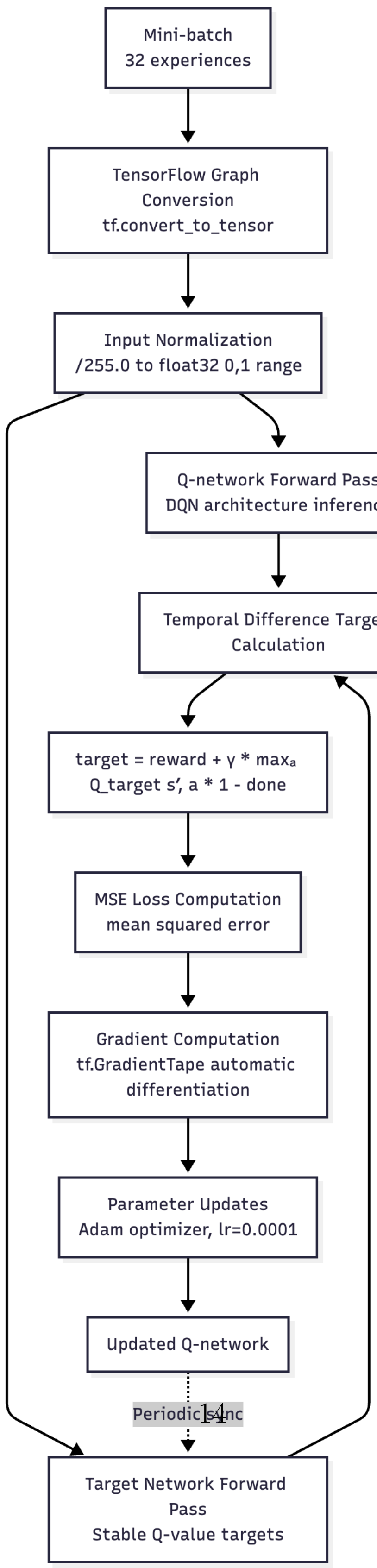

Figure 7: Learning Process

The learning process uses a decoupled architecture where the target network provides stable Q-value targets while the main Q-network undergoes continuous optimization. The `@tf.function` decorator compiles the training step into an optimized computational graph, significantly improving performance through operation fusion and efficient memory management.

4.4 System Synchronization and Monitoring Subsystem

This subsystem ensures system stability, maintains training consistency, and provides real-time performance monitoring:

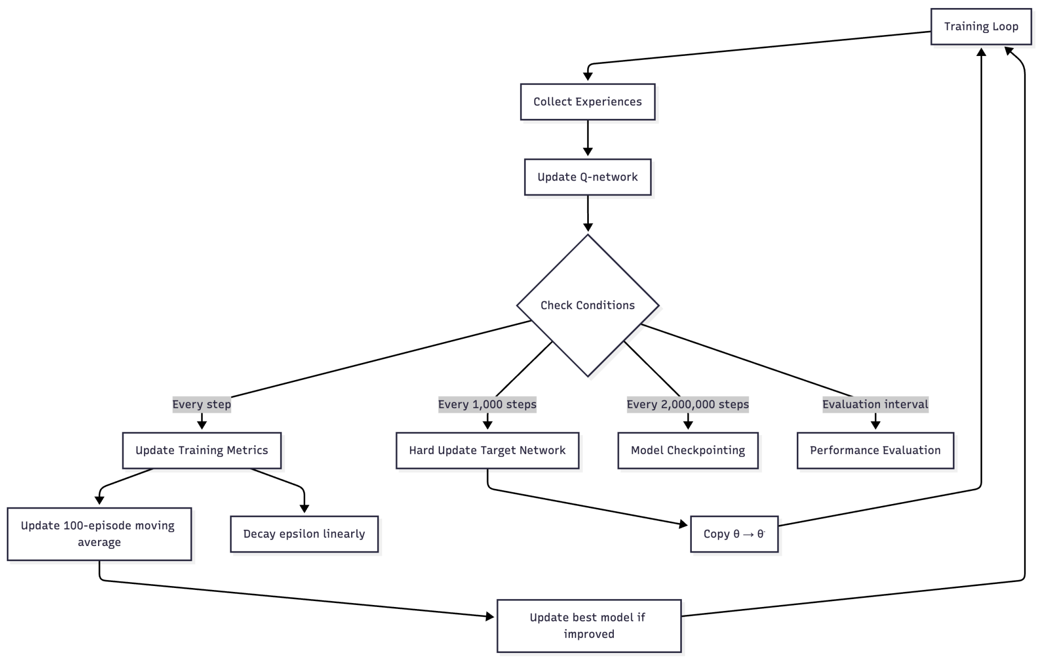

Figure 8: Synchronization And Monitorig Process

The synchronization mechanism implements a hard update strategy where the target network is completely overwritten with the Q-network's parameters at fixed intervals. This approach, combined with comprehensive logging and model checkpointing, ensures training stability and enables recovery from suboptimal convergence.

4.5 Integrated Data Flow Coordination

These subsystems operate concurrently in a carefully orchestrated workflow:

1. **Continuous Data Collection:** The environment interaction subsystem runs continuously, generating new experiences regardless of learning frequency.
2. **Asynchronous Learning:** The learning subsystem activates every 4 environment steps, processing random mini-batches from the replay buffer.
3. **Periodic Synchronization:** The target network updates every 1,000 steps, providing stable temporal difference targets.
4. **Systematic Evaluation:** Performance monitoring occurs at fixed intervals, guiding model selection and hyperparameter tuning.

This architecture enables efficient resource utilization by decoupling experience collection from learning updates, while maintaining training stability through experience replay and target network separation. The system progressively refines the agent's policy through this coordinated cycle of exploration, experience storage, systematic learning, and performance evaluation.

5 Results and Analysis

5.1 Training Progress and Performance

Due to computational constraints and the extensive time required to train reinforcement learning (RL) agents on a CPU, the training process for the DQN agent was terminated prematurely at episode 6,600, instead of the planned 10,000 episodes. Despite this early termination, the agent exhibited meaningful learning progress and demonstrated consistent improvements in performance.

At the point of termination, the key training statistics were as follows:

Metric	**Value**
Total Environment Steps	4,997,673
Episodes Completed	6,600
Average 100-Episode Reward	71.66
Exploration Rate (ϵ)	0.100 (minimum value)
Average 100-Episode Length	1,187.05 steps

Table 1: Training statistics at termination (episode 6,600)

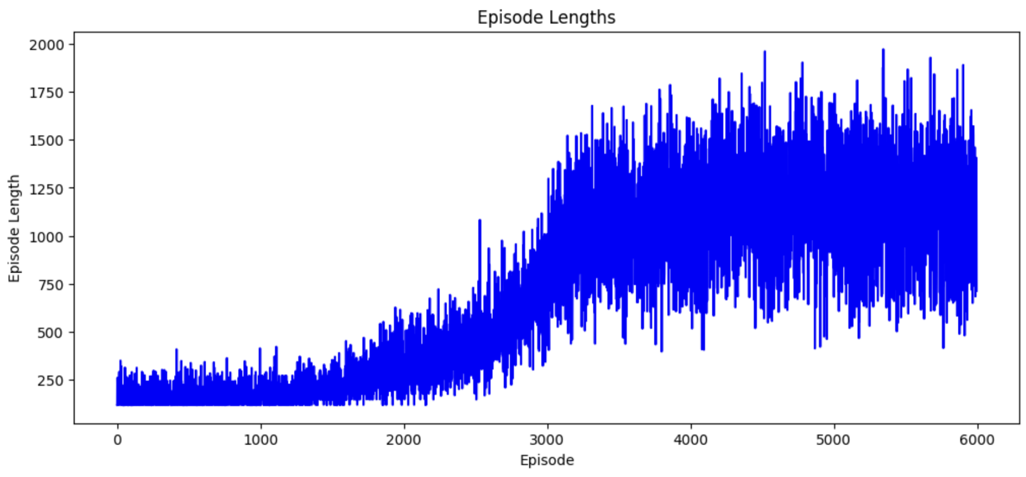

Figure 9: Episode Length Over Episodes

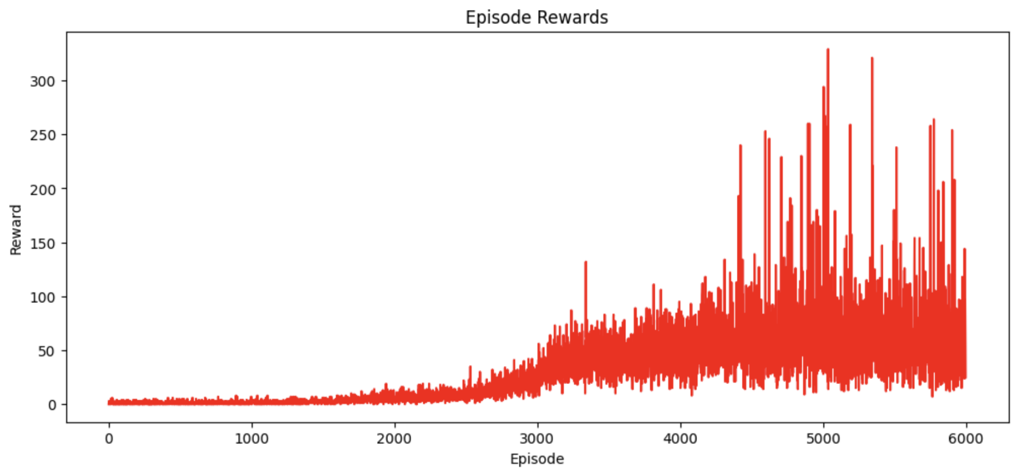

Figure 10: Total Episodic Rewards Over Episodes

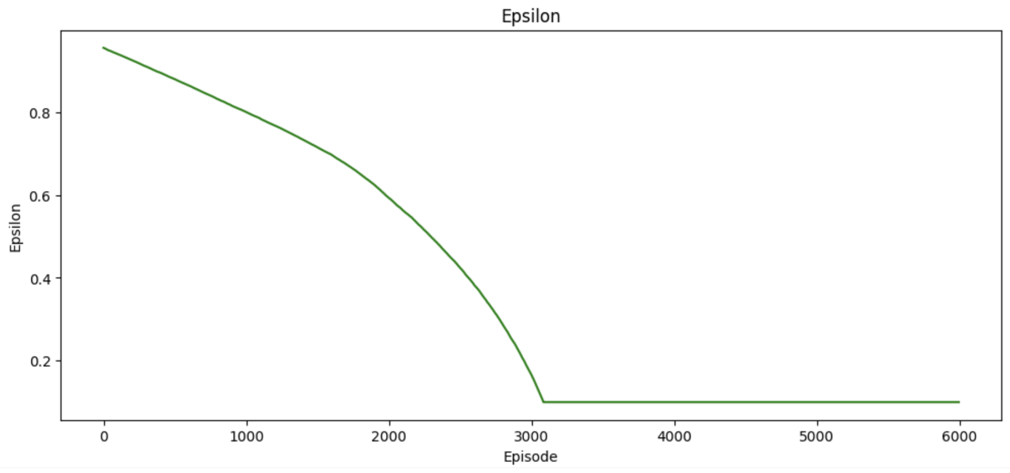

Figure 11: Epsilon Decay Over Episodes

5.2 Learning Curve and Performance Trajectory

5.2.1 Reward Progression

The reward curve shows a clear upward trajectory over the course of training, indicating that the agent gradually learned more effective gameplay strategies. Early in training (episodes 0–2000), rewards remained near zero, which corresponds to random exploration. As training advanced (episodes 2000–4000), rewards began to increase significantly, reflecting the emergence of basic policy learning. Beyond episode 4000, the agent achieved intermittent spikes in performance, with several episodes exceeding rewards of 250–300, signifying strong but somewhat unstable learned behavior.

The final average 100-episode reward of 71.66 indicates substantial improvement from the initial random policy and confirms successful learning of the game's dynamics.

5.2.2 Exploration vs. Exploitation

The epsilon decay curve reveals a smooth decline from $\epsilon = 1.0$ to $\epsilon = 0.1$, where it stabilized after approximately 3,500 episodes. This shows the gradual shift from exploration (random action selection) to exploitation (policy-driven action selection). By the time ϵ reached 0.1, the agent was predominantly leveraging learned knowledge to make decisions, with only occasional exploratory actions to refine its understanding.

5.2.3 Episode Length Analysis

The episode length curve closely correlates with reward trends. Longer episode durations reflect the agent's improved ability to survive longer and avoid losing the game early. The average 100-episode length of 1,187 steps near the end of training signifies that the agent had learned to sustain gameplay effectively, a clear indicator of improved policy stability and state-action understanding.

5.3 Learned Capabilities and Behavioral Insights

Through training, the agent acquired several key gameplay competencies:

- **Ball Tracking:** Developed the ability to accurately follow the ball's trajectory across the screen.
- **Paddle Positioning:** Learned effective paddle control to intercept the ball reliably.
- **Basic Strategy Formation:** Displayed emerging strategies for targeting and clearing brick structures.

- **Life Management:** Demonstrated adaptive responses to avoid losing lives, indicating awareness of game penalties.

While the performance had not yet plateaued, the steady upward trend suggested that continued training could further refine the agent's strategies and potentially achieve near-human performance levels.

5.4 Computational Constraints and Efficiency Challenges

The training process was heavily influenced by hardware limitations, as the experiments were conducted without GPU acceleration. Major challenges included:

- **CPU-Only Execution:** Each episode required approximately 5–10 minutes, resulting in extremely long training durations.
- **High Computational Load:** Real-time frame preprocessing, stacking operations, and forward/backward passes in the neural network contributed to significant overhead.
- **Large State Representation:** The 84844 input state size demanded extensive matrix computations for each time step, further slowing down the process.

These limitations collectively restricted the total number of trainable episodes and prevented full convergence within the allotted time.

6 Conclusion

Despite being prematurely halted at 6,600 episodes, the DQN agent demonstrated clear signs of effective learning and policy improvement in the Breakout environment. The rising reward curve, extended episode lengths, and stabilized epsilon decay collectively indicate successful training progress toward an optimized gameplay strategy.

Key takeaways include:

- The agent effectively transitioned from random to strategic behavior.

- The performance trends suggest convergence toward an optimal policy, though not yet fully achieved due to early termination.
- Further trainingpreferably on GPUwould likely enhance stability, reward consistency, and overall performance.

In summary, this project successfully implemented and validated the Deep Q-Network (DQN) algorithm for Breakout under limited computational resources, proving the feasibility of meaningful RL training even with hardware constraints.