CS525: Reinforcement Learning

Project 3 - Report

Student: Sumukh, Porwal

1 Introduction

This report presents an exploration of several advanced Deep Q-Network (DQN) architectures and configurations for training an agent to play the Breakout game. The objective was to identify the most effective DQN variant by conducting a series of experiments with different network architectures, hyperparameters, and loss functions. In the end, Double Deep Q-Network (DDQN) emerged as the most successful approach, demonstrating significant stability, enhanced training convergence, and performance efficiency. This report details the experiments conducted, describes the neural network architecture selected, outlines the hyperparameters tuned, and explains the choice of loss function, along with a rationale for why each aspect contributed positively to the agent's performance.

2 Experiments with DQN Variants

7 DQN variants were explored in this project, each designed to address certain challenges in the traditional DQN framework:

1. **DQN (Deep Q-Network)**: The baseline method that learns the Q-value of each state-action pair. While effective, it suffers from overestimation bias, which can lead to suboptimal policies.

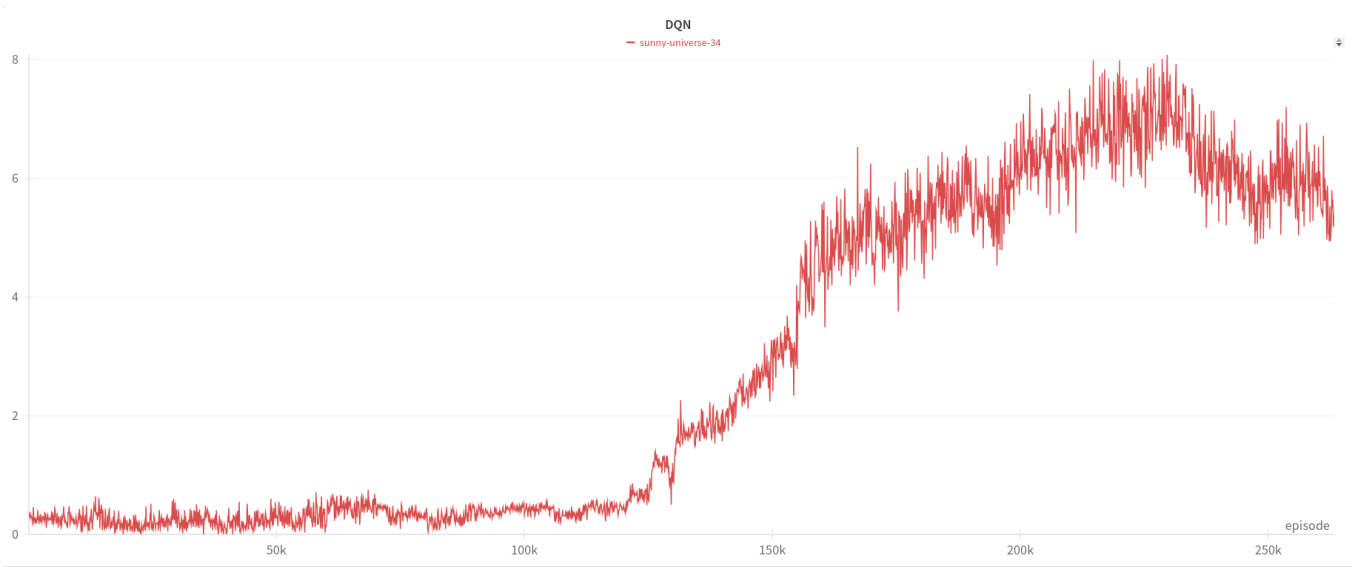

Figure 1: DQN Learning Curve

2. **Dueling DQN**: This architecture separates the estimation of the value function from the advantage function, making it more resilient to non-optimal actions by focusing on the value of states themselves. However, it did not significantly improve stability in the Breakout environment, likely due to the limited benefit of distinguishing state and action values in this context.

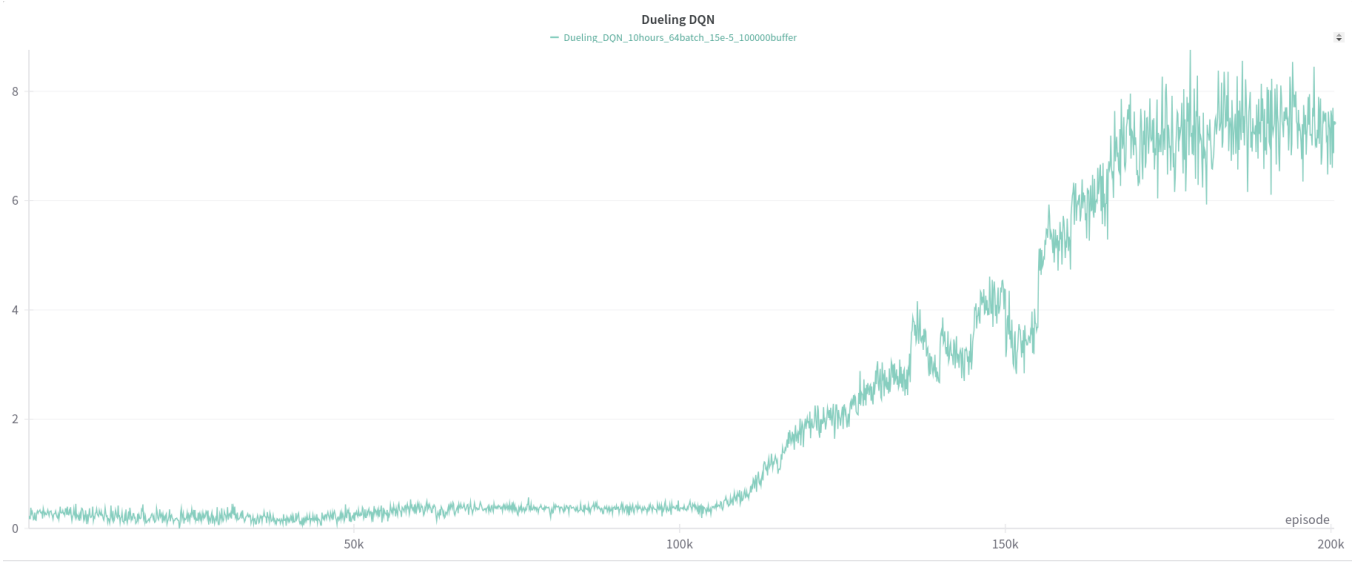

Figure 2: Dueling DQN Learning Curve

3. **Prioritized DQN**: Prioritized experience replay prioritizes important transitions, which theoretically allows the agent to focus on high-impact experiences. In practice, though, this added complexity without a clear performance gain, possibly due to the sufficient sample diversity in a standard replay buffer given the game's dynamic environment.

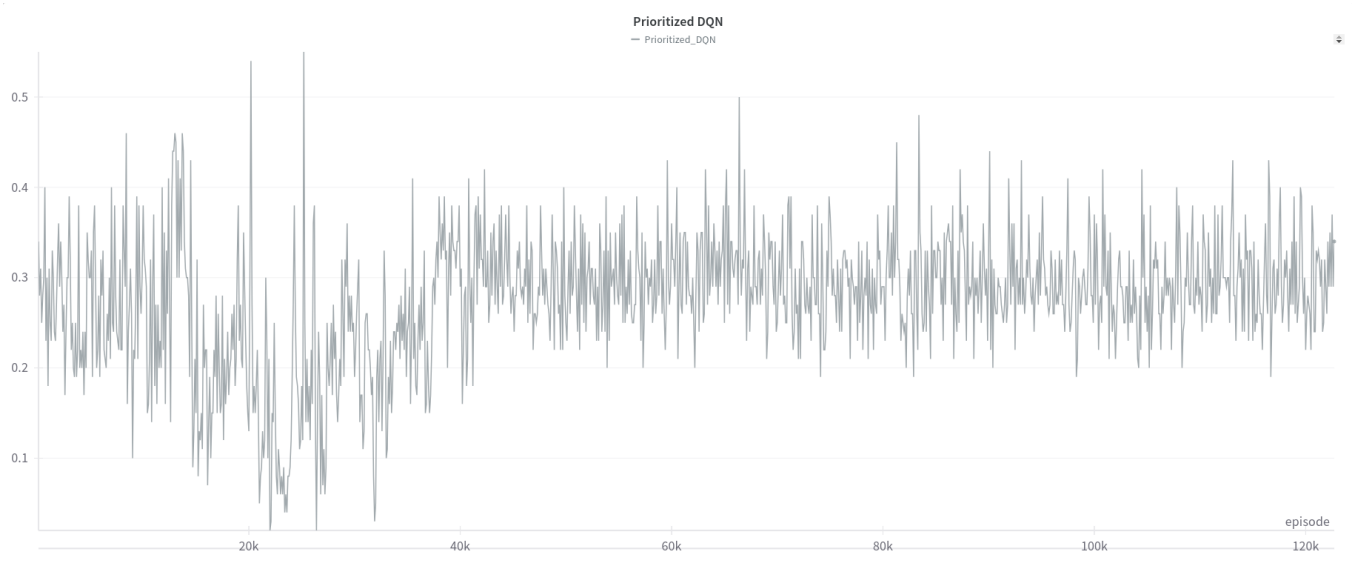

Figure 3: Prioritized DQN Learning Curve

4. **Double DQN (DDQN)**: This variant addresses overestimation by decoupling action selection and evaluation, reducing the positive bias in Q-value estimation and improving learning stability. In our experiments, this approach excelled, as it helped the agent learn more balanced Q-values, leading to more consistent policies.

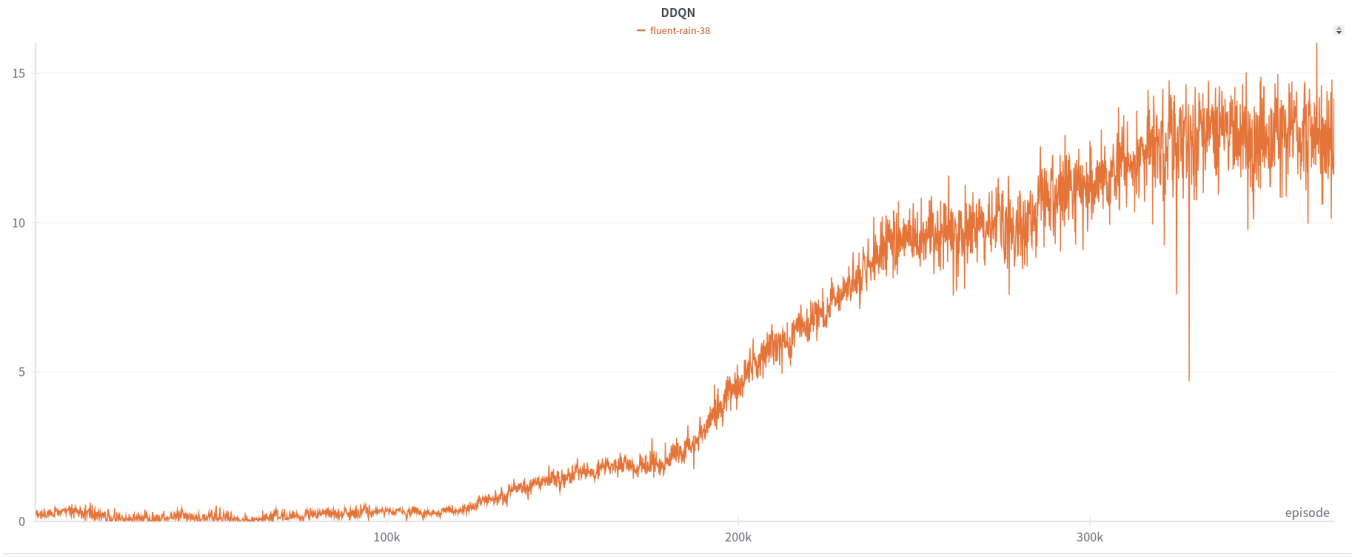

Figure 4: DDQN Learning Curve

5. **Prioritized DDQN**: By combining prioritized experience replay with DDQN, this variant aims to enhance learning by focusing on influential experiences while mitigating overestimation. However, it introduced more complexity without significant gains in stability or speed for this environment.

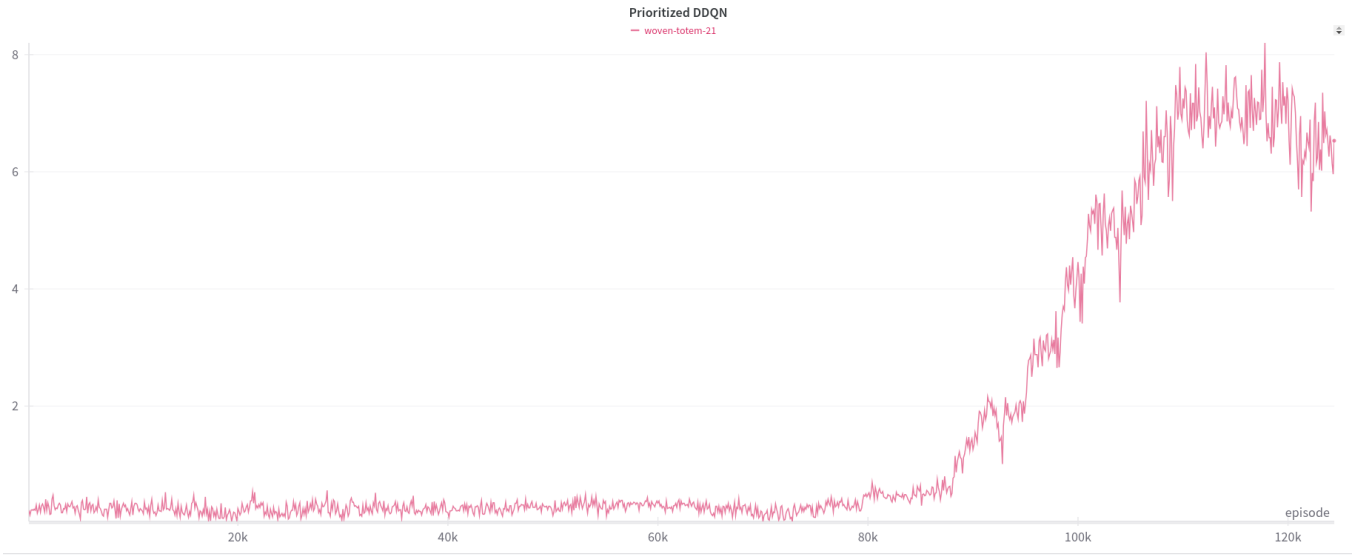

Figure 5: Prioritized DDQN Learning Curve

6. **Multistep DQN**: Extending DQN with multistep returns theoretically captures longer-term dependencies, but for Breakout, it sometimes led to greater instability due to the challenge of balancing short-term and long-term gains.

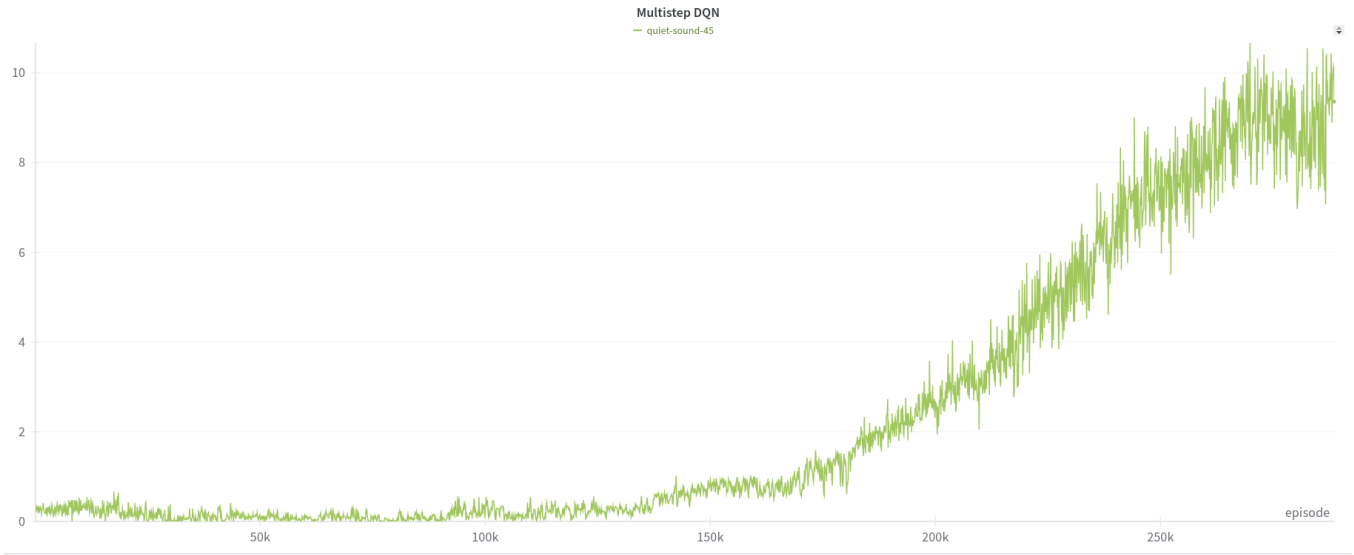

Figure 6: Multistep DQN Learning Curve

7. **Multistep DDQN**: This method combines multistep returns with DDQN, aiming to stabilize long-term learning while reducing overestimation bias. While beneficial in theory, it introduced some overfitting to specific action sequences in Breakout.

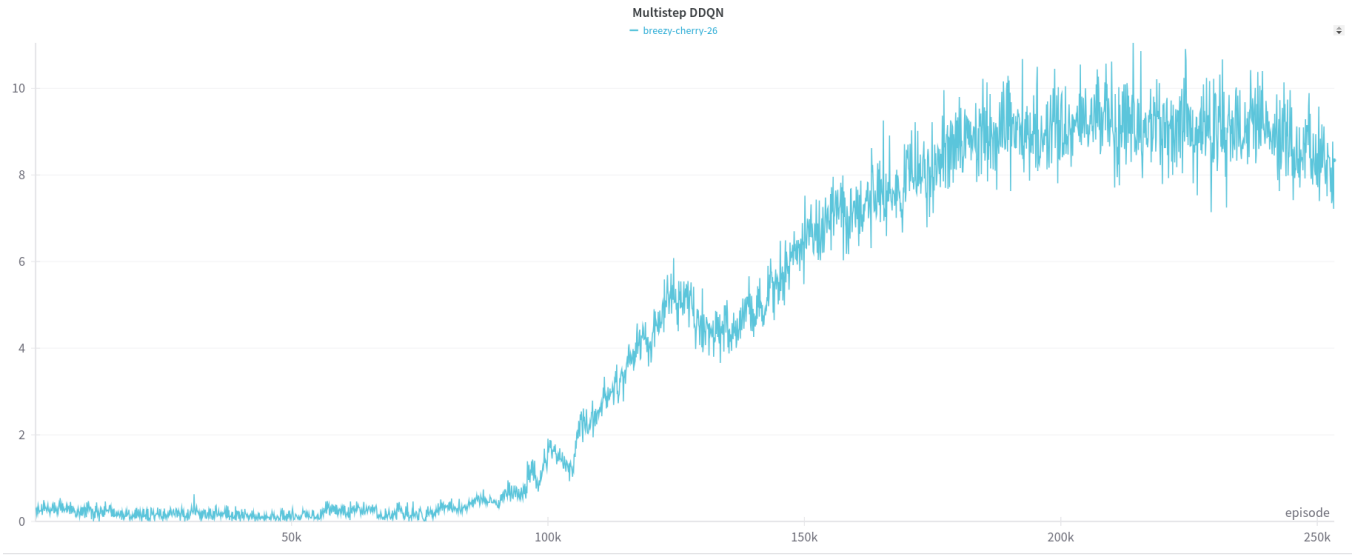

Figure 7: Multistep DDQN Learning Curve

Ultimately, the Double DQN (DDQN) approach achieved the best balance between computational efficiency and learning effectiveness. DDQN's decoupling of action selection and evaluation is particularly beneficial in environments like Breakout, where Q-value overestimation can lead to aggressive or risk-prone policies that underperform.

3 Neural Network Architecture

In this project, several neural network architectures were evaluated to identify the structure that could best capture the essential spatial and temporal features necessary for success in the Breakout environment. After testing various configurations, a convolutional neural network (CNN) with three convolutional layers followed by two fully connected layers emerged as the optimal architecture.

3.1 Selected Architecture

The selected architecture consists of three convolutional layers, followed by two fully connected layers. Each convolutional layer was designed to progressively abstract spatial information from the game screen, capturing both the position and movement of objects (like the ball and paddle), which are critical for decision-making in Breakout.

- The first convolutional layer has 32 filters, an 8x8 kernel, and a stride of 4. This configuration provides a large receptive field that captures the primary movement of game objects across the screen, allowing the network to quickly identify general spatial layouts.
- The second convolutional layer has 64 filters, a 4x4 kernel, and a stride of 2. This layer further refines spatial information, focusing on smaller details within the game frame.
- The third convolutional layer also has 64 filters with a 3x3 kernel and a stride of 1, enabling the network to capture fine-grained movement and object interactions crucial for learning optimal gameplay strategies.

Following the convolutional layers, the extracted features are flattened and passed to a fully connected layer with 512 neurons and a ReLU activation function, which prepares the data for the final decision-making layer. This is followed by an output layer with a number of units equal to the action space size in the game. The chosen architecture allowed the agent to balance computational efficiency and accuracy, making it effective in capturing relevant patterns without overfitting or slowing down learning due to excessive complexity.

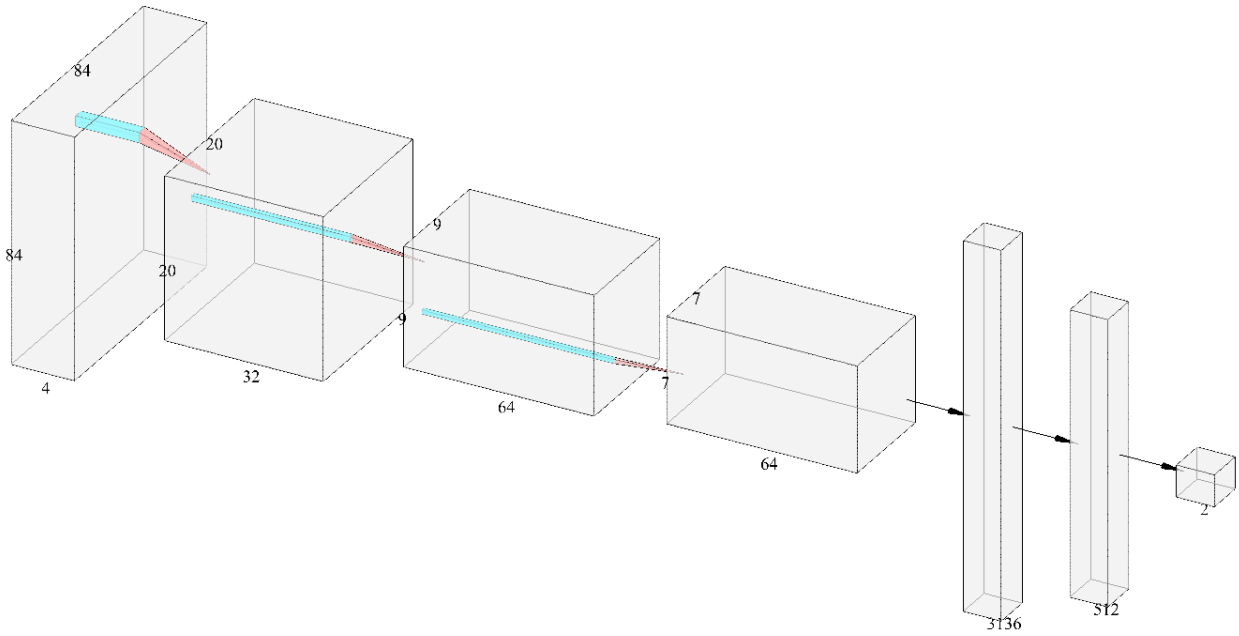

Figure 8: Neural Network Architecture

3.2 Alternative Architectures

Several alternative architectures were evaluated, but these either lacked the representational power to capture critical features or introduced unnecessary complexity, which slowed training and sometimes led to overfitting. Below are the key alternatives that were tested and reasons why they were ultimately outperformed by the selected architecture:

1. **Simpler CNN with Fewer Filters and Layers**: An architecture with only two convolutional layers (one with 16 filters and the other with 32 filters) was tested initially. While this model was computationally efficient, it failed to capture the intricate movement of the ball and paddle, resulting in a weaker ability to anticipate necessary actions. The agent trained with this simpler network often underperformed, especially in scenarios requiring quick reactions to multiple game objects.

2. **Deeper CNN with Additional Convolutional and Fully Connected Layers**: To explore the impact of greater network depth, a model with five convolutional layers and three fully connected layers was also tested. While this architecture had significant capacity, it led to prolonged training times and higher susceptibility to overfitting. The deeper network captured high-frequency details in the input, but these additional layers did not enhance the agent's understanding of the broader spatial dynamics in Breakout, leading to negligible performance improvements relative to the increase in complexity.

3.3 Why the Chosen Architecture Worked Best

The selected architecture with three convolutional layers and two fully connected layers struck an effective balance between complexity and representational power. It was able to capture essential spatial information without incurring the computational costs associated with deeper or more complex models. The sequential use of progressively smaller kernels in the convolutional layers allowed the agent to efficiently identify both global and local features, such as ball trajectory and paddle positioning, which are crucial for playing Breakout effectively.

Simpler architectures lacked the depth necessary to learn complex strategies, while more complex architectures introduced overfitting risks and longer training times. The final selected architecture maximized learning efficiency and provided robust generalization, making it the optimal choice for this application.

4 Hyperparameter Tuning

Key hyperparameters were tuned to optimize training stability and agent performance. The final values chosen were:

- **Discount Factor (GAMMA)**: Set to 0.99, this high discount factor prioritizes long-term rewards, which is essential in Breakout where future outcomes often depend on cumulative actions. Lower GAMMA values would overly favor immediate gains, reducing the agent's ability to learn prolonged strategies.
- **Batch Size (BATCH_SIZE)**: A batch size of 1024 was found optimal for stable training. Larger batches increase the sample diversity within each gradient update, allowing the agent to learn more robustly from each experience. Smaller batch sizes led to noisier updates, which made training less stable.
- **Replay Buffer Size (BUFFER_SIZE)**: With a buffer size of 50,000, this configuration retained sufficient historical data to improve learning stability, providing the agent with a diverse set of experiences without overwhelming memory capacity. Smaller buffer sizes led to insufficient variability, while larger buffers introduced unnecessary memory usage without added benefit.
- **Epsilon Parameters for Exploration Decay**:
  - **Initial Epsilon (EPSILON)**: Set to 1, encouraging full exploration at the start to expose the agent to a variety of states.
  - **Final Epsilon (EPSILON_END)**: Set to 0.025, allowing some degree of exploration even in later training stages to avoid overfitting to specific strategies.
  - **Decay Steps (DECAY_EPSILON_AFTER)**: Epsilon begins decaying after 5000 steps, enabling a gradual shift from exploration to exploitation, which proved essential in letting the agent discover effective strategies without premature convergence.
- **Learning Rate (LEARNING_RATE)**: A learning rate of 0.0005 was optimal for convergence speed and stability. Lower values led to slower learning, while higher values resulted in unstable training, as the agent would oscillate around suboptimal strategies.

The above hyperparameter choices provided a balanced learning process that allowed the agent to explore adequately, learn from a variety of experiences, and optimize its policy without becoming overly deterministic.

5 Loss Function and Optimizer

Training an agent to perform well in reinforcement learning environments like Breakout requires a robust loss function and optimizer. For this project, the Huber loss, also known as Smooth L1 loss, was chosen as the loss function, while the Adam optimizer was selected to adjust the network weights. These choices were made after evaluating several alternatives, with each aspect contributing significantly to the improved stability and performance of the agent.

5.1 Huber Loss

The Huber loss function, or Smooth L1 loss, is a combination of Mean Squared Error (MSE) and Mean Absolute Error (MAE), depending on the magnitude of the prediction error. For smaller errors, it behaves like MSE, which is beneficial for minimizing error magnitudes, while for larger errors, it transitions to MAE, reducing sensitivity to outliers. This hybrid approach provides two major advantages for reinforcement learning tasks:

- **Stability**: In reinforcement learning, where large or highly variable gradients can cause instability, Huber loss helps stabilize training by reducing the impact of large temporal difference errors, which are common when the agent initially learns to play the game.
- **Robustness to Outliers**: During training, the agent may occasionally encounter unexpected large errors (e.g., when an action results in a sudden drop in reward). Huber loss, with its linear component for large errors, mitigates the impact of these outliers, leading to more consistent training dynamics.

Alternative loss functions, such as Mean Squared Error (MSE) or Mean Absolute Error (MAE), were tested but did not perform as effectively. MSE tends to be sensitive to large errors, which can lead to high variance in updates, making it less stable in reinforcement learning contexts. MAE, on the other hand, provides stability but can be slower to converge as it does not emphasize smaller errors, which limits fine-tuning during training. Huber loss provided an ideal balance, combining the benefits of both MSE and MAE to improve both stability and learning speed.

5.2 Adam Optimizer

The Adam optimizer, which combines the benefits of Adaptive Gradient Algorithm (AdaGrad) and Root Mean Square Propagation (RMSProp), was selected for this project due to its efficiency in handling sparse and noisy gradients, which are common in reinforcement learning tasks.

- **Adaptive Learning Rates**: Adam uses adaptive learning rates based on moment estimates of both the gradient and its square, which allows it to maintain optimal step sizes across different parameter updates. This adaptability proved beneficial in stabilizing the agent's training, as it helped prevent both overshooting and slow convergence, which are common issues in reinforcement learning.
- **Bias Correction**: Adam includes a bias correction step that reduces the initial bias in the moment estimates. This is particularly useful in the early stages of training when the gradients are not yet well-calibrated, leading to more accurate updates and faster learning.
- **Combining Momentum with Gradient Descent**: By incorporating momentum, Adam helps the agent navigate the optimization landscape more effectively, especially in cases with noisy gradients, allowing it to learn more efficiently in the complex Breakout environment.

Alternative optimizers, such as Stochastic Gradient Descent (SGD), RMSProp, and AdaGrad, were evaluated but did not yield the same level of performance. SGD, while computationally simple, struggled with noisy gradients, making it difficult for the agent to converge to optimal solutions efficiently. RMSProp, which adapts learning rates based on recent gradient magnitudes, did provide some stability but lacked the bias correction and full momentum benefits of Adam. AdaGrad's learning rate decay led to slower convergence over time, especially in later training stages when more fine-tuning was required.

5.3 Summary

The combination of Huber loss and the Adam optimizer proved to be the most effective setup for this project. Huber loss provided a balanced approach to error minimization, reducing sensitivity to outliers while maintaining the ability to fine-tune. Meanwhile, Adam's adaptive learning rates, momentum, and bias correction made it well-suited for navigating the complex reward landscape in Breakout, leading to faster convergence and more consistent performance. This combination ultimately enabled the agent to learn more efficiently and achieve superior gameplay compared to other configurations.

6 Conclusion

Through rigorous experimentation with various DQN variants, neural network architectures, and hyperparameters, it was concluded that **Double DQN (DDQN)** performed best, thanks to its ability to minimize Q-value overestimation and stabilize learning. The chosen network architecture effectively captured key spatial and temporal features of the Breakout game, while the tuned hyperparameters provided a balance between exploration and exploitation. Finally, Huber Loss contributed to the model's robustness, enabling efficient learning in a dynamic environment. These choices, collectively, supported the development of a highly competent Breakout-playing agent.

7 Results

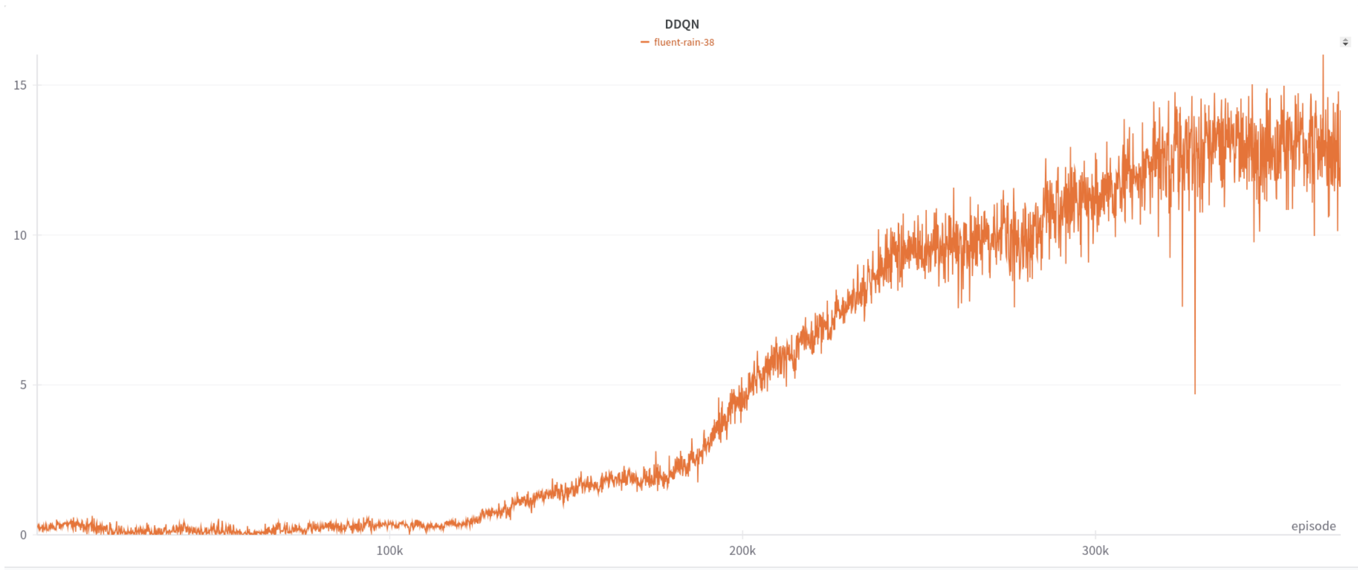

Figure 9: Learning curve of DQN

```
  test = torch.load('final dqn model reward 14.5.pth')
100%|████████████████████████████████████████| 100/100 [03:30<00:00,  2.11s/it]
Run 100 episodes for 5 lives each
Mean: 309.54
rewards [159.0, 392.0, 392.0, 392.0, 159.0, 265.0, 159.0, 392.0, 338.0, 265.0, 338.0, 338.0, 392.0, 265.0, 338.0, 159.0, 392.0, 392.0, 338.0, 392.0, 392.0, 392.0, 392.0, 338.0, 392.0, 159.0, 338
.0, 338.0, 392.0, 265.0, 338.0, 338.0, 338.0, 338.0, 265.0, 159.0, 338.0, 159.0, 392.0, 265.0, 265.0, 265.0, 265.0, 159.0, 265.0, 159.0, 392.0, 392.0, 265.0, 338.0, 392.0, 392.0, 338.0, 338.0, 3
38.0, 392.0, 392.0, 159.0, 265.0, 392.0, 338.0, 159.0, 338.0, 392.0, 265.0, 392.0, 159.0, 392.0, 265.0, 392.0, 338.0, 159.0, 159.0, 392.0, 392.0, 265.0, 338.0, 159.0, 159.0, 265.0, 392.0, 392.0,
 338.0, 159.0, 159.0, 392.0, 392.0, 338.0, 265.0, 338.0, 338.0, 338.0, 392.0, 392.0, 338.0, 265.0, 338.0, 265.0, 265.0, 392.0]
running time 210.86489963531494
running time: 216.46270537376404
```

Figure 10: Test results

A final test result with a reward of ***309.54*** was obtained.