

# Deep Q-Learning for Flappy Bird: Implementation and Evaluation

Rusu Andrei-Dudu, Zloteanu Mircea

January 4, 2025

## Abstract

This report presents the implementation and evaluation of a Deep Q-Learning (DQN) agent trained to play Flappy Bird using PyTorch and Gymnasium. The report details the neural network architecture, hyperparameter settings, and experimental results. Various training runs were conducted to evaluate the performance of the agent under different configurations. The best-performing agent achieved a maximum score of **763** across multiple runs.

## 1 Introduction

Flappy Bird is a challenging and fast-paced game, making it an ideal testbed for reinforcement learning algorithms. This project leverages the Deep Q-Learning algorithm with experience replay and a convolutional neural network (CNN) to approximate Q-values. The goal is to train an agent capable of achieving a high score consistently.

## 2 Methodology

### 2.1 Agent Architecture

The DQN agent consists of two main components:

1. **Neural Network (NN):** The network is designed to approximate Q-values for each action using convolutional and fully connected layers:
  - **Input Layer:** Grayscale image of size  $84 \times 84$ .
  - **Convolutional Layers:**
    - Layer 1:  $1 \rightarrow 32$  channels, kernel size  $3 \times 3$ , stride 1, padding 1.
    - Layer 2:  $32 \rightarrow 64$  channels, kernel size  $3 \times 3$ , stride 1, padding 1.
    - Layer 3:  $64 \rightarrow 128$  channels, kernel size  $3 \times 3$ , stride 1, padding 1.
    - Layer 4:  $128 \rightarrow 256$  channels, kernel size  $3 \times 3$ , stride 1, padding 1.
  - **Fully Connected Layers:**
    - Flatten layer: Output from convolutional layers flattened to  $256 \times 5 \times 5$ .
    - Dense Layer 1:  $256 \times 5 \times 5 \rightarrow 512$  units.

- Output Layer:  $512 \rightarrow 2$  (Q-values for actions).
2. **Replay Buffer:** A fixed-size replay buffer stores past experiences for sampling during training.

## 2.2 Hyperparameters

The training hyperparameters used for the DQN agent are summarized in Table 1.

Parameter	Value
Learning Rate ( $\eta$ )	$1 \times 10^{-5}$
Discount Factor ( $\gamma$ )	0.99
Epsilon ( $\epsilon$ )	$0.1 \rightarrow 0.0001$ (decay 0.995)
Batch Size	32
Replay Buffer Capacity	50,000
Training Start Threshold	5,000 experiences
Number of Epochs	2,000

Table 1: Hyperparameters for training the DQN agent.

## 2.3 Explanation of Hyperparameters

The hyperparameters used in training the DQN agent play a crucial role in determining its performance. Below is a detailed explanation of each parameter:

- **Learning Rate ( $\eta$ ):** A small learning rate of  $1 \times 10^{-5}$  ensures that the updates to the neural network weights are gradual, reducing the risk of overshooting the optimal values during training.
- **Discount Factor ( $\gamma$ ):** The discount factor of 0.99 prioritizes long-term rewards, helping the agent to plan strategies over multiple steps instead of focusing solely on immediate gains.
- **Epsilon ( $\epsilon$ ):** Initially set to 0.1, epsilon facilitates exploration by encouraging the agent to take random actions. Over time, it decays to 0.0001 with a decay rate of 0.995, shifting the agent’s behavior towards exploitation of its learned policy.
- **Batch Size:** A batch size of 32 ensures computational efficiency while maintaining sufficient diversity in the training samples, improving the robustness of updates to the neural network.
- **Replay Buffer Capacity:** The replay buffer holds up to 50,000 experiences, providing a diverse set of past interactions for training. This helps to reduce temporal correlations in the data, stabilizing the learning process.
- **Training Threshold:** The agent begins training only after 5,000 experiences are collected in the replay buffer. This ensures a diverse set of data is available, preventing premature updates that could hinder learning.

- **Number of Epochs:** Training the agent over 2,000 epochs allows sufficient iterations for the neural network to learn and adapt to the environment, achieving stable performance.

These carefully chosen hyperparameters ensure that the agent learns effectively while balancing exploration and exploitation. The use of a replay buffer and batch sampling further contributes to stable and efficient training.

## 2.4 Preprocessing

The input images undergo the following preprocessing steps:

1. Crop the bottom 110 pixels to remove irrelevant information.
2. Convert the cropped image to grayscale.
3. Resize to  $84 \times 84$  pixels.
4. Normalize pixel values to  $[0, 1]$  and convert to tensors.

## 2.5 Training Procedure

The training process consists of:

1. Epsilon-greedy action selection.
2. Experience replay with randomly sampled mini-batches.
3. Q-value updates using the Bellman equation.
4. Saving model checkpoints every 100 epochs.

# 3 Implementation Details

## 3.1 Q-Learning Algorithm

The Q-learning algorithm uses the Bellman equation to update the Q-values:

$$Q(s, a) \leftarrow Q(s, a) + \alpha \left( r + \gamma \max_{a'} Q(s', a') - Q(s, a) \right)$$

where:

- $s$  and  $s'$  are the current and next states.
- $a$  and  $a'$  are the current and next actions.
- $r$  is the reward received after taking action  $a$  in state  $s$ .
- $\alpha$  is the learning rate.
- $\gamma$  is the discount factor.

In the implementation, the target Q-values are computed as follows:

Listing 1: Target Q-values Calculation

```
def compute_target_q_values(self, q_values, actions, rewards,
                             next_values, dones):
    """Compute the target Q-values for the Bellman equation"""
    target_values = q_values.clone()
    for i in range(len(target_values)):
        target_values[i][actions[i]] = (rewards[i]
                                         + self.gamma * torch.max(
                                             next_values[i]) * (1 - dones
                                                                  [i]))
    return target_values
```

#### Explanation:

- The function takes in the current Q-values, actions, rewards, next state Q-values, and a terminal state indicator (`dones`).
- For each action  $a_i$ , the target Q-value is calculated using the Bellman equation.
- The term  $(1 - \text{dones}[i])$  ensures that if the next state is terminal ( $\text{done} = 1$ ), the discounted future reward is ignored.
- The target Q-value replaces the predicted Q-value for the corresponding action in the batch.

This implementation ensures the agent learns effectively from both immediate and delayed rewards.

## 3.2 Neural Network Implementation

The neural network used in this project was implemented using PyTorch. It is a Convolutional Neural Network (CNN) specifically designed to process pixel-based inputs from the game environment and output Q-values for the possible actions.

### 3.2.1 Architecture

The architecture of the neural network consists of the following layers:

- **Convolutional Layers:** Three convolutional layers are used to extract spatial features from the pixel-based input. These layers apply filters to capture patterns such as edges, textures, and movement within the game frames.
- **Fully Connected Layers:** Following the convolutional layers, fully connected layers map the extracted features to Q-values for each possible action. This allows the network to estimate the expected rewards for each action given the current state.
- **Activation Functions:** Rectified Linear Units (ReLU) are applied after each layer to introduce non-linearity and allow the network to model complex patterns.

The input to the network is a preprocessed image from the game environment, resized to a fixed resolution and converted to grayscale. This reduces the computational complexity and focuses on essential visual information.

### 3.2.2 Training Setup

The training process is guided by the following choices:

- **Optimizer:** The Adam optimizer is employed for efficient gradient descent. It dynamically adjusts the learning rate for each parameter based on the first and second moments of the gradients.
- **Loss Function:** The Mean Squared Error (MSE) loss is used to minimize the difference between the predicted Q-values and the target Q-values derived from the Bellman equation. This encourages the model to make more accurate predictions over time.
- **Replay Buffer:** A replay buffer is used to store past experiences (state, action, reward, next state, done). During training, batches of experiences are sampled randomly to break temporal correlations and stabilize training.
- **Batch Size:** A fixed batch size is used for training to balance memory usage and computational efficiency.
- **Epsilon Decay:** An epsilon-greedy strategy is applied during training to encourage exploration of the game environment. The value of epsilon decays over time to prioritize exploitation of learned strategies as training progresses.

### 3.2.3 Why Convolutional Neural Networks?

CNNs are chosen for this implementation because they are particularly effective at extracting spatial features from images. By convolving filters across input frames, the network can learn to recognize important visual cues, such as the position of obstacles and the bird’s location, which are critical for decision-making in the game.

## 4 Experiments and Results

### 4.1 Experimentation

Multiple training runs were conducted with varying seeds. The performance was evaluated based on:

- **Game Scores:** Average and maximum scores achieved in the Flappy Bird environment.

### 4.2 Results

The performance of the agent is summarized in Table 2.

## 5 Conclusion

The DQN agent successfully learned to play Flappy Bird, achieving a maximum score of **763**. Training on pixel data proved challenging but effective after extensive preprocessing and hyperparameter tuning. Future improvements include exploring advanced RL techniques like Double DQN or Prioritized Experience Replay.

Epoch	Average Score	Best Score
1000	1	3
2000	10	27
2020	25	73
2030	43	98
2070	50	115
2120	55	122
2230	70	159
2290	85	219
2340	75	219
2550	78	225
2585	84	308
2600	93	308
2620	102	323
2670	114	371
2870	133	408
2900	170	578
3040	211	578
3100	233	675
3250	267	763

Table 2: The agent’s performance during training, showing both the average score (indicating consistency) and the best score (representing peak performance) at various epochs.

```
Epoch: 200, Loss: 0.06342114508152008, Score: 77, Best score: 763
101 0.50000 0.0001
```

## References

1. Neural Networks Reference
2. Gymnasium Documentation
3. Flappy Bird Environment Repository
4. PyTorch Documentation