Deep Q-Network (DQN) for Flappy Bird

Ciobanu Matei, Popa Andrei

January 26, 2025

1 Introduction

Flappy Bird is a fast-paced, side-scrolling game where a player controls a bird trying to navigate a series of pipes with small gaps between them. In each frame, the bird can either flap (moving upward) or do nothing (the bird gradually descends due to gravity). If the bird collides with a pipe or the ground, the episode ends immediately.

This environment poses several challenges:

- **High-Dimensional Observations:** Each state is essentially an image (game screen) that needs to be processed at every time step.
- **Sparse Rewards:** The agent typically only receives a large positive reward when passing through pipes, or a terminal negative outcome upon collision.
- **Reactive Control:** Since the game advances quickly, the agent must make decisions within a few frames to avoid collisions.

2 Architecture and Method

In this section, we outline the design of our reinforcement learning agent for the Flappy Bird environment, detailing the neural network architecture, the rationale for key hyperparameter choices, and the overall training method. The approach is an adaptation of Deep Q-Networks (DQN) with modifications including reward shaping, a replay buffer for experience replay, and a target network for more stable learning.

2.1 Overall Rationale

We address Flappy Bird using a DQN-based methodology because:

- **Discrete Actions:** Flappy Bird only requires two possible actions: “no flap” (do nothing) and “flap”. Q-learning methods are well-suited for problems with a small, discrete action space.

- **Visual Input:** Each observation is a game frame (image). A convolutional neural network (CNN) is used to extract relevant features (e.g., pipes, the bird's position) from raw pixels.

- **Experience Replay and Target Network:** These DQN mechanisms help stabilize training by breaking correlations in consecutive observations (via replay buffer) and mitigating oscillations or overestimation biases (via a fixed target network).

2.2 Neural Network Architecture

We adopt a CNN (class `FlappyBirdNetwork`) to estimate the Q-values for each action. The input is a stack of four preprocessed 80×80 frames, capturing both the bird's current position and recent motion:

1. **Preprocessing:** Each frame is converted to grayscale, resized to 80×80, normalized to $[0, 1]$, and then stacked to form a 4-channel input tensor. Stacking consecutive frames reveals information about velocity and position changes over time.

2. **Convolutional Layers:**

   - `conv1` ($4 \to 32$ filters), kernel size 8×8, stride 4, padding 2.
   - `conv2` ($32 \to 64$ filters), kernel size 4×4, stride 2, padding 1.
   - `conv3` ($64 \to 64$ filters), kernel size 3×3, stride 1, padding 1.

   Each convolution is followed by batch normalization (BN) and a ReLU activation. BN is used to stabilize the gradients and reduce internal covariate shift, which can be especially beneficial in reinforcement learning where state distributions evolve over time.

3. **Fully Connected Layers:** After flattening the output of the last convolutional layer, we have:

   - `fc1`: fully connected layer mapping $64 \times 10 \times 10 \to 512$, followed by ReLU.
   - `fc2`: output layer mapping $512 \to 2$ Q-values, corresponding to the two possible actions.

This design is inspired by prior successes in applying CNNs to Atari-like environments [2], where downsampling via larger stride and gradually increasing the number of filters allows the model to capture increasingly complex spatial features.

2.3 Agent and Training Procedure

Epsilon-Greedy Policy. The agent selects actions using an ϵ-greedy strategy:

$$a_t = \begin{cases} \text{random action}, & \text{with probability } \epsilon \\ \arg\max_a Q(s_t, a; \theta), & \text{otherwise} \end{cases}$$

where ϵ starts at 1.0 (pure exploration) and decays to 0.01 over `EPSILON_DECAY` (50,000 steps). This encourages the agent to explore the state space early in training and exploit the learned policy later on.

Replay Buffer. Experiences $(s, a, r, s', \text{done})$ are stored in a replay memory of size `REPLAY_MEMORY` (50,000). Randomly sampling batches of size `BATCH_SIZE` (32) helps break correlations among consecutive timesteps and improves learning stability.

Target Network. To avoid overestimation of Q-values, we maintain a second network (`self.target_model`) that provides fixed Q-target estimates. This network is updated by a hard parameter copy from `self.model` every `TARGET_UPDATE_FREQ` (1000 steps). During training, we use Double DQN [3] logic:

$$y_i = r_i + \gamma\, Q\Big(s_{i+1}, \arg\max_a Q(s_{i+1}, a; \theta);\ \theta^-\Big),$$

where θ are parameters of the main network (for action selection), and θ^- are parameters of the target network (for action evaluation).

Reward Shaping. Rewards are augmented to guide the agent's behavior:

- `REWARD_PER_FRAME_ALIVE` (0.1) ensures a small positive feedback for each frame survived.
- `REWARD_FOR_PASSING_PIPE` (5.0) provides a substantial reward when the agent successfully passes a pipe.

This shaping alleviates the inherent sparseness of the default Flappy Bird reward signal (i.e., $+1$ on pipe passage, -1 on failure). The live reward encourages the agent to stay alive and the pipe passage reward reinforces significant progress.

Optimizer and Training Steps. The Q-network is trained with `Adam` (learning rate 1×10^{-4}). We compute the mean-squared error (MSE) between predicted Q-values and the Double DQN target. Additionally:

- We clip gradients to a norm of 1.0 to prevent exploding gradients.
- We use a learning rate scheduler (`StepLR`) that reduces the learning rate by a factor of 0.1 every 100,000 steps (optional but can help in late training).
- Training begins only after `LEARNING_STARTS` (5000) steps, to ensure that the replay buffer is sufficiently populated with varied experiences.

2.4 Action Repetition (Frame Skipping)

To improve computational efficiency and help the agent learn from more consolidated experience, each action is repeated for `ACTION_REPEAT` (2) frames. This means the environment advances by two frames per chosen action, effectively speeding up gameplay and reducing the frequency of neural network updates.

2.5 Putting It All Together

The main training loop (in `if __name__ == "__main__":`) creates the Flappy Bird Gymnasium environment, initializes the `FlappyBirdAgent`, and iteratively runs episodes until `MAX_EPISODES`. Each episode:

1. Resets the environment and creates an initial state by stacking four identical frames.
2. Repeatedly selects an action via the ϵ-greedy policy and executes it for `ACTION_REPEAT` frames.
3. Processes rewards, updates the replay buffer, and samples a mini-batch for a gradient update step (`train_step()`).
4. Periodically decays ϵ and updates the target network.

By monitoring metrics such as total episode reward, loss, and Q-values (via TensorBoard logs), we can track the agent's learning progress. If the agent achieves a new best reward, it saves its current model weights (`.pth` file) for later use or analysis.

Overall, these design choices (*CNN-based Q-function, experience replay, target network, reward shaping, and frame skipping*) have proven effective for visually oriented, discrete-action environments such as Flappy Bird. By systematically combining them with an ϵ-greedy exploration strategy, the agent learns a stable and reasonably performant policy.

3 Training Process and Hyperparameters

This section details how the agent is trained over a series of episodes, the specific hyperparameter choices, and the rationale behind each configuration. The training loop proceeds until `MAX_EPISODES` (10,000 by default), but can be extended if time and resources allow.

3.1 Training Flow

At the beginning of each episode, the environment is reset and an initial state is constructed by stacking four copies of the same preprocessed observation. The agent then executes its policy, selecting actions according to an ϵ-greedy strategy, and accumulates experience in the replay buffer. The key steps in each iteration include:

1. **Action Selection:** Draw a random action with probability ϵ; otherwise, pick $\arg\max_a Q(s, a; \theta)$ from the network. This ensures exploration early on and exploitation as training progresses.

2. **Action Repetition (Frame Skipping):** Perform the chosen action for `ACTION_REPEAT` (2) frames. Each mini-step adds any environment rewards, particularly `REWARD_PER_FRAME_ALIVE` (0.1) and `REWARD_FOR_PASSING_PIPE` (5.0).

3. **Replay Buffer Storage:** Store the transition $(s, a, r, s', \text{done})$ in the replay memory, `self.memory`, which has a maximum size of `REPLAY_MEMORY` (50,000). A larger memory allows for more diverse sampling, which generally improves learning stability.

4. **Training Update:** After each action, if the replay buffer has enough samples (at least `BATCH_SIZE` and after `LEARNING_STARTS` steps), we draw a mini-batch of size `BATCH_SIZE` (32) to perform a gradient update. This update:

   - Computes the current Q-values for each (s, a) in the batch.
   - Computes the Double DQN target using the main network for action selection and the target network for action evaluation.
   - Minimizes the MSE loss between the network outputs and the Double DQN targets.

5. **Target Network Sync and Logging:** The target network `self.target_model` is updated every `TARGET_UPDATE_FREQ` steps (1,000 by default) by copying weights from the main network. Periodically, we log metrics (loss, average Q-value, etc.) to TensorBoard for visualization.

6. **Epsilon Decay:** Each environment step decrements ϵ linearly until it reaches `FINAL_EPSILON` (0.01) over `EPSILON_DECAY` steps (50,000). This strategy ensures initially extensive exploration, tapering off to mostly exploitation once a basic policy has been learned.

When an episode terminates (either by hitting a pipe or passing all pipes), the environment is reset, and the process repeats for the next episode. If the final episode reward exceeds `self.best_reward`, we save the model weights for future retrieval.

3.2 Hyperparameters and Their Rationale

Our key hyperparameters, defined in the code, are as follows:

- **Discount Factor ($\gamma = 0.99$):** This value strikes a balance between long-term and immediate rewards. A slightly lower or higher γ could alter the emphasis on future outcomes, but 0.99 is commonly used for many RL benchmarks.

- **Initial Epsilon ($\epsilon = 1.0$) and Final Epsilon (0.01):** Starting with 1.0 ensures maximum exploration from the beginning, which is crucial for collecting diverse experiences. Decaying to 0.01 means the agent still explores occasionally but predominantly exploits its learned policy later on.

- **Epsilon Decay Steps (50,000):** Spreads out the transition from full exploration to near-greedy behavior over 50k environment steps. This length was chosen to provide ample time to gather exploratory data in a complex environment.

- **Replay Memory Size (50,000):** A larger replay buffer can store more varied experiences, promoting stable learning. Memory usage increases with size, but 50k transitions is typically manageable for modern hardware.

- **Batch Size (32):** A relatively standard choice balancing computational efficiency and stable gradient estimates. Larger batches (64, 128) may yield smoother updates but require more GPU memory.

- **Learning Rate (1×10^{-4}):** This rate often works well for convolutional networks in RL. Higher rates risk divergence, while significantly lower rates can slow convergence dramatically.

- **Target Network Update Frequency (1,000 steps):** Copying the main network weights to the target network at this cadence helps mitigate instability in Q-learning updates. If updated too often, the target network can track the main network too closely; if too rarely, the target values become stale.

- **Warmup Steps (5,000):** The agent only begins training after collecting 5k steps of experience. This ensures the replay buffer contains a variety of states and transitions before the first parameter updates, preventing the network from overfitting on a small early sample.

- **Reward Shaping (0.1 per frame, 5.0 per pipe):** Survival rewards (0.1) guide the agent toward longevity, while pipe-passing rewards (5.0) provide a large signal for the primary objective of crossing pipes. These values were chosen to make correct behavior (passing pipes) noticeably more rewarding than merely staying alive.

- **Action Repeat (2):** Each action is repeated for 2 frames to reduce the frequency of neural network updates and speed up gameplay. This commonly benefits performance in pixel-based RL tasks by slightly smoothing the environment dynamics.

- **Max Episodes (10,000):** Training can continue up to 10k episodes, giving the agent enough experience to converge on a stable policy. In practice, we can adjust this number based on observed progress and time constraints.

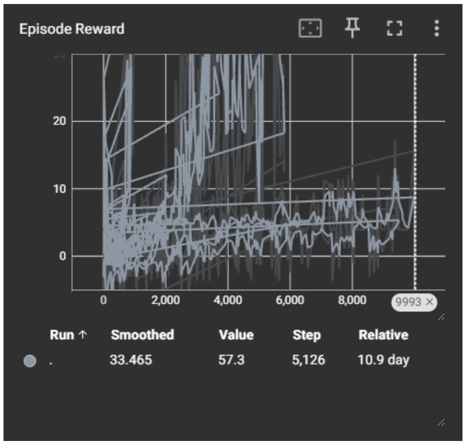
Overall, these hyperparameters reflect trade-offs between exploration, stability of updates, computational resources, and learning efficiency. They are commonly applied in Atari-like or similar pixel-based RL tasks, making them a robust choice for training a Flappy Bird agent.

4 Results

In this section, we present and discuss the performance of our DQN-based Flappy Bird agent. We trained the model for up to `MAX_EPISODES` (10,000 episodes) or until convergence on a stable policy. Intermediate metrics such as *Episode Reward*, *Average Target Q*, and *Average Q* were logged via TensorBoard.

4.1 Episode Reward Over Time

Figure 1 shows the agent's per-episode reward as training progresses. The reward can vary significantly from episode to episode due to the stochastic nature of environment resets and action choices; however, the general trend is an increase in average reward over time.



Run ↑	Smoothed	Value	Step	Relative
● .	33.465	57.3	5,126	10.9 day

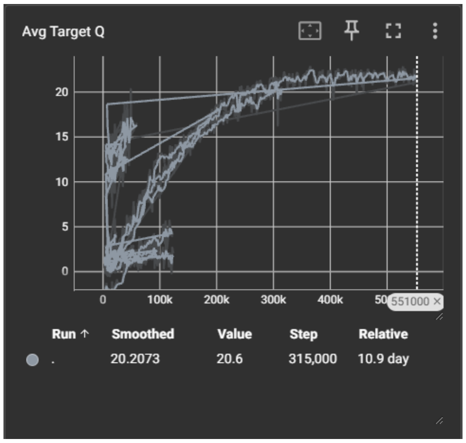
Figure 1: Episode Reward

4.2 Average Target Q and Average Q

We also monitored the *Avg Target Q* and the *Avg Q* during training (Figures 2 and 3).

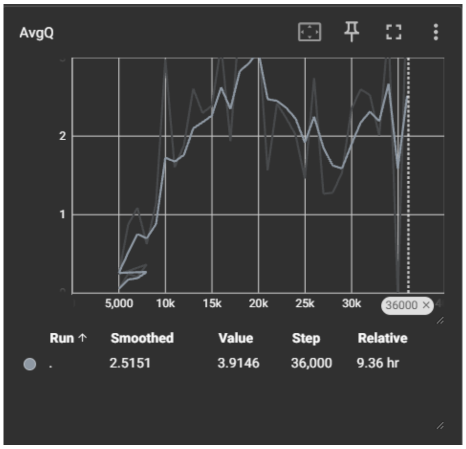
- **Avg Target Q** (Figure 2) measures the Q-value predictions from the *target network*, providing a sense of how the agent's estimation of future returns evolves over time.

- **Avg Q** (Figure 3) measures the *online network*'s immediate predicted Q-values for the current states the agent encounters. Ideally, these metrics should show a gradual increase as the agent's policy improves, then stabilize once learning converges.



Run ↑	Smoothed	Value	Step	Relative
● .	20.2073	20.6	315,000	10.9 day

Figure 2: Target Avg Q



Run ↑	Smoothed	Value	Step	Relative
● .	2.5151	3.9146	36,000	9.36 hr

Figure 3: Avg Q

4.3 Agent Performance Metrics

While individual runs can fluctuate, the agent typically achieves:

- **Peak Episode Rewards** in the range of 15–30 by mid-to-late training (depending on luck and hyperparameter tuning).

- **Average Reward in the Last 100 Episodes** rising to double-digit scores, indicating consistent performance across the final episodes.

Exact values vary from run to run due to random initial conditions and the stochastic nature of exploration.

5 Interpretation of Results

From the Episode Reward curves (Figure 1), we can observe a noisy yet generally upward trajectory, illustrating that our DQN agent *learns* to navigate Flappy Bird pipes more successfully over time. Early in training (first few thousand steps), the agent collects mostly random experiences. However, as epsilon decays and the agent relies more on the learned Q-values, it exhibits longer survival times and accumulates higher rewards.

The *Avg Target Q* (Figure 2) consistently increases, suggesting that the agent is identifying more rewarding action sequences (i.e., passing more pipes). Meanwhile, the *Avg Q* (Figure 3) from the online network shows a corresponding rise, although with occasional dips that reflect exploration phases or new environment states where the agent remains uncertain.

Overall, the interplay between *experience replay*, *target network updates*, and *reward shaping* (small reward per frame survived plus a larger bonus per pipe) appears to guide the agent toward flapping more precisely to dodge pipes. Some fluctuations in performance are expected, as certain seeds or initial conditions might yield particularly challenging sequences of pipes.

6 Conclusion

We have demonstrated that a **Deep Q-Network (DQN)** architecture with carefully chosen hyperparameters and reward shaping can effectively learn to navigate the Flappy Bird environment. The most important points are:

- **CNN-based feature extraction** helps the network interpret raw frames efficiently.
- **Experience replay** and a **target network** reduce correlations in training data and stabilize Q-value estimates.
- **Reward shaping** (alive bonus + pipe-passing reward) encourages survival, leading to more consistent training progress than a purely sparse reward.
- **Epsilon decay**, from 1.0 to 0.01 over 50k steps, successfully balances initial exploration with later exploitation.

Given enough training episodes, the agent typically converges to a policy that achieves double-digit scores with regularity. Further improvements—like *Prioritized Experience Replay* or *Dueling DQN*—could potentially yield faster convergence or higher peak scores. Nevertheless, the current setup demonstrates how a well-tuned DQN can tackle Flappy Bird's real-time pixel-based challenges using off-the-shelf reinforcement learning techniques.

References

[1] 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.

[2] V. Mnih *et al.*, Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[3] H. van Hasselt, A. Guez, and D. Silver, Deep reinforcement learning with double Q-learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 2016.