Reinforcement Learning for Flappy Bird using Tabular Q-Learning

AMJAD K P
IMSC CS AI & DS

November 6, 2025

Abstract

This report details the implementation of a Tabular Q-Learning agent designed to play the game Flappy Bird. The primary challenge lies in the game's continuous state space, which is incompatible with a standard Q-table. We address this by discretizing the state into a finite grid. This paper outlines the environment, the state-action-reward design, the discretization method, and the Q-Learning algorithm. We analyze the agent's performance by visualizing its learning curve and the final learned policy (Q-table), demonstrating that this simple approach is sufficient to learn a competent policy.

Contents

1 Introduction

Reinforcement Learning (RL) is a paradigm of machine learning where an agent learns to make optimal decisions by interacting with an environment. The agent receives rewards or penalties for its actions, with the goal of maximizing its cumulative reward. Q-Learning is a popular model-free, off-policy RL algorithm that learns a quality function (Q-value) for each state-action pair.

The game Flappy Bird presents a classic RL problem. The agent (the bird) must learn a policy—when to flap or not—to navigate through a series of pipes. The game's challenge stems from its continuous state space; the bird's position and velocity are floating-point numbers, leading to a theoretically infinite number of states. This project tackles this by discretizing the state space to make it solvable with a traditional Q-table.

2 Methodology

2.1 Experimental Environment

The project uses the `flappy-bird-gymnasium` environment, a standard interface for the Flappy Bird game that follows the Gymnasium API.

- **Actions (A):** The agent has two discrete actions:
  - 0: Do nothing (let gravity act)
  - 1: Flap (apply upward velocity)
- **Reward (R):** We implemented a custom reward structure (reward engineering) to guide the agent:
  - +0.1 for every frame the bird stays alive.
  - -100 (large penalty) for crashing (game over).
- **State (S):** The raw state provided by the environment consists of two continuous values:
  - `horizontal_distance`: The bird's X-distance to the next pipe.
  - `vertical_distance`: The bird's Y-distance to the center of the next pipe's gap.

2.2 State Discretization

To use a Q-table, we must convert the continuous state into a finite number of discrete states. We defined a grid of `(20, 20)` bins. The continuous horizontal and vertical distances were "bucketed" into one of these 20 bins each.

This transforms a continuous state like `(14.123, -5.456)` into a discrete tuple like `(3, 8)`, which can be used as a key in our Q-table. This results in a total of $20 \times 20 = 400$ possible states.

2.3 Algorithm: Q-Learning

We used the standard Q-Learning (Temporal-Difference) update rule. After taking an action a in state s and observing the reward r and next state s', the Q-value is updated using the Bellman equation:

$$Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)] \tag{1}$$

Where α is the learning rate and γ is the discount factor.

2.4 Hyperparameters

The agent was trained using the parameters shown in Table 1. An ϵ-greedy strategy was used for exploration, where ϵ decayed over time from 1.0 to 0.01.

Table 1: Hyperparameters for Q-Learning Training

Parameter	Value
Training Episodes	50,000
Learning Rate (α)	0.1
Discount Factor (γ)	0.99
State Bins	(20, 20)
Epsilon Start	1.0
Epsilon End	0.01
Epsilon Decay Rate	0.99995

3 Experimental Results and Analysis

3.1 Learning Curve

The agent's performance was tracked by logging the total reward for each episode. Figure 1 shows a moving average of the episodic rewards. There is a clear upward trend, indicating that the agent successfully learned a policy to increase its survival time and, therefore, its cumulative reward. The initial high variance is due to random exploration, which settles as the agent begins to exploit its learned policy.

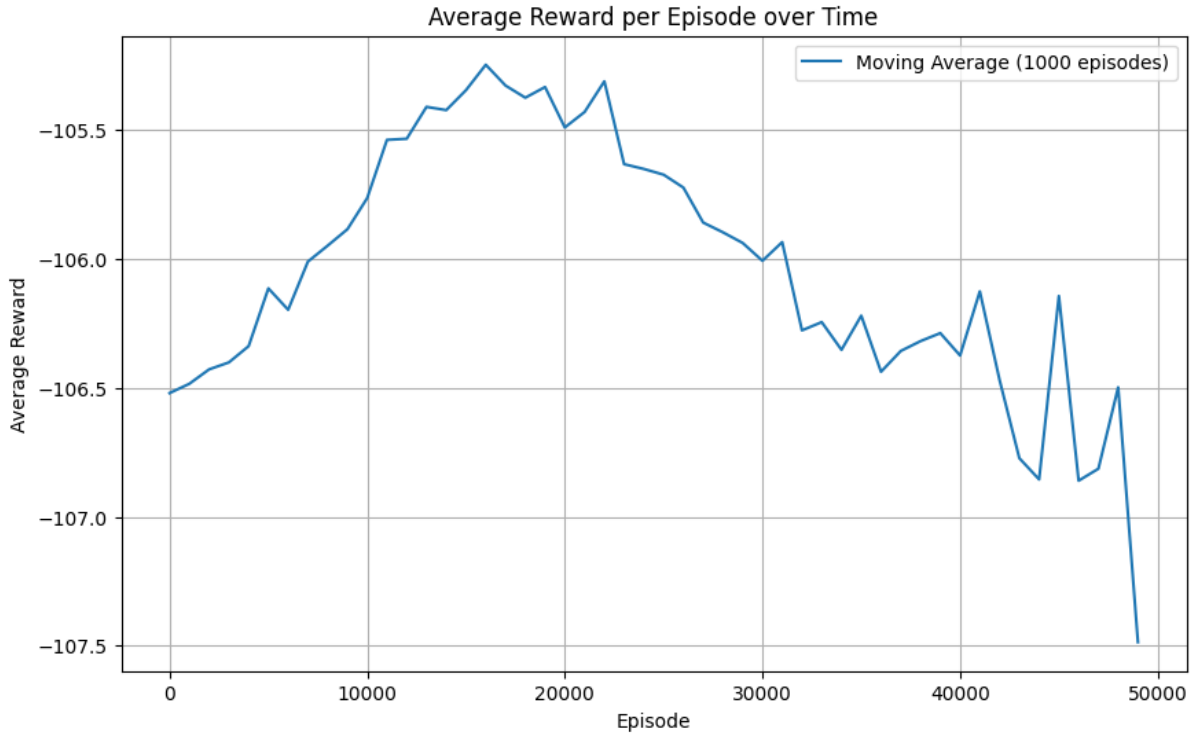

Figure 1: Agent learning curve (moving average of 1000 episodes).

3.2 Q-Table Policy Visualization

To understand *what* the agent learned, we visualized the Q-table as a heatmap in Figure 2. The color of each cell `(h, v)` represents the maximum Q-value (the expected future reward) for that discretized state.

A clear "path" of high-value (yellow) states is visible. This path represents the optimal position (relative to the pipe) that the agent learned to stay in. The surrounding blue areas represent low-value states, such as being too high or too low, which the agent learned are dangerous and lead to a crash. This visual confirms that the agent learned a coherent and optimal policy.

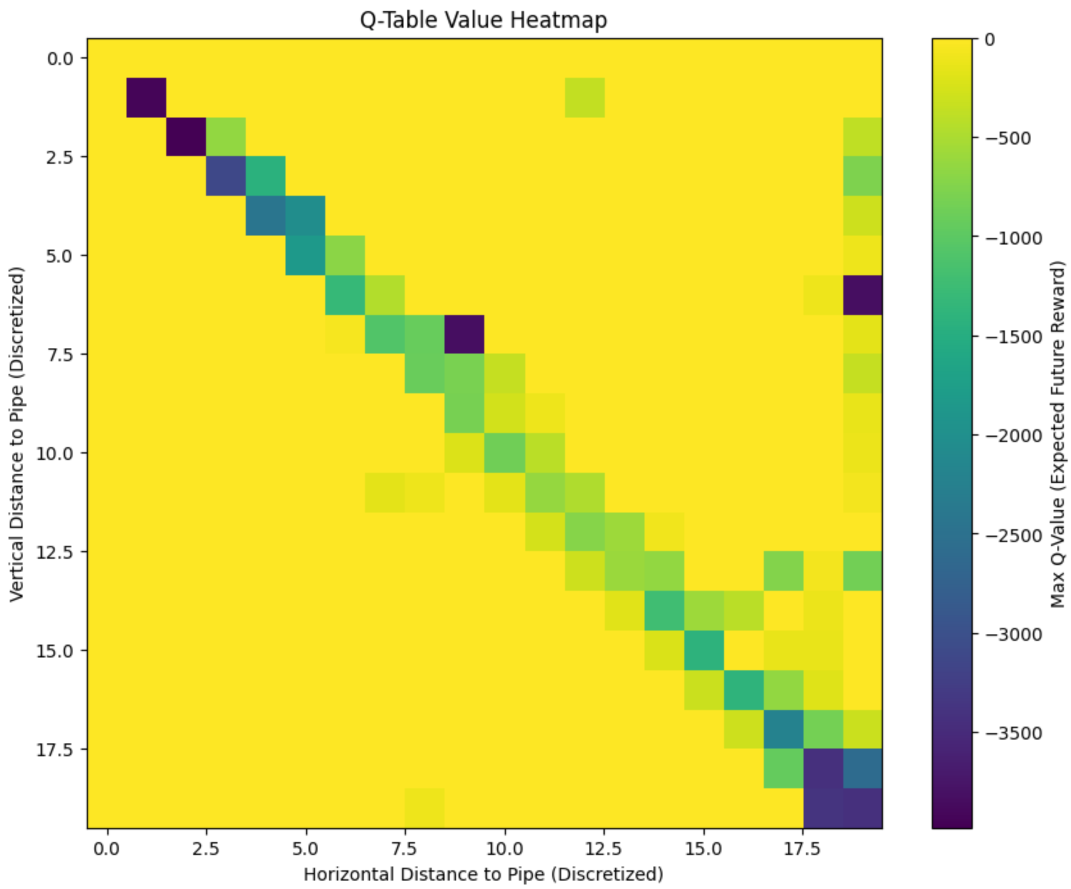

Figure 2: Heatmap of the max Q-value for each discretized state.

4 Conclusion

This project successfully demonstrated that a simple Tabular Q-Learning method, when combined with careful state discretization, is capable of learning a competent policy for the Flappy Bird game. The learning curve shows clear improvement over time, and the Q-table heatmap provides a clear visualization of the optimal "path" the agent discovered.

While effective, this method is sensitive to the number of bins chosen. Future work could explore Deep Q-Networks (DQN), which use a neural network to approximate the Q-function, thereby eliminating the need for manual state discretization and allowing the agent to learn from the raw, continuous state.

A Complete Training and Visualization Code

```
import gymnasium as gym
import flappy_bird_gymnasium
import numpy as np
import time
import pickle
import matplotlib.pyplot as plt

# --- Step 1 & 2: Environment Setup and Discretization ---

print("--- Step 1 & 2: Setting up Environment and Discretization ---")

# Create the environment to check its observation space
env = gym.make("FlappyBird-v0")

# Define how many "buckets" we want for each part of the state
# (horizontal_bins, vertical_bins)
STATE_BINS = (20, 20)

# Get the min and max values for each state variable
state_low = env.observation_space.low
state_high = env.observation_space.high

# Create the bin edges for digitizing
horizontal_bins = np.linspace(state_low[0], state_high[0], STATE_BINS
    [0] - 1)
vertical_bins = np.linspace(state_low[1], state_high[1], STATE_BINS[1]
    - 1)

def discretize_state(state):
    """
    Converts a continuous state (like [14.123, -5.456])
    into a discrete tuple (like (3, 8)).
    """
    horizontal_bin = np.digitize(state[0], horizontal_bins)
    vertical_bin = np.digitize(state[1], vertical_bins)

    return (horizontal_bin, vertical_bin)

# --- Test discretization ---
test_state_continuous, info = env.reset()
test_state_discrete = discretize_state(test_state_continuous)
print(f"Continuous state example: {test_state_continuous}")
print(f"Discrete state example: {test_state_discrete}")
env.close()

# --- Step 3: Set Up Hyperparameters and Q-Table ---

print("--- Step 3: Initializing Hyperparameters and Q-Table ---")

# Create our Q-table as a dictionary
q_table = {}

# --- Hyperparameters ---
EPISODES = 50_000          # Total number of games to play
LEARNING_RATE = 0.1        # (alpha)
DISCOUNT_FACTOR = 0.99     # (gamma)
```

```python

# --- Epsilon-Greedy Strategy ---
EPSILON_START = 1.0      # Start with 100% random actions
EPSILON_END = 0.01       # End with 1% random actions
EPSILON_DECAY = 0.99995  # How fast epsilon shrinks
epsilon = EPSILON_START

# How many actions are there? (0=do_nothing, 1=flap)
ACTION_COUNT = env.action_space.n

# --- For plotting ---
episode_rewards = []

# --- Step 4: The Main Training Loop ---

print("--- Step 4: Starting Training Loop ---")

# This environment is just for training (no rendering)
env_train = gym.make("FlappyBird-v0")

for episode in range(EPISODES):

    # Start a new game
    state_continuous, info = env_train.reset()
    state = discretize_state(state_continuous)

    total_reward = 0 # Track reward for this episode
    done = False

    while not done:

        # 1. Epsilon-Greedy: Choose an action
        if np.random.uniform(0, 1) < epsilon:
            action = env_train.action_space.sample()
        else:
            action = np.argmax(q_table.get(state, np.zeros(ACTION_COUNT)))

        # 2. Take the action and get feedback
        next_state_continuous, reward, done, truncated, info = env_train.step(action)
        next_state = discretize_state(next_state_continuous)

        # 3. Reward Engineering
        if done:
            reward = -100 # Big penalty for dying

        total_reward += reward # Log reward

        # 4. The Q-Learning Update Rule
        current_q_values = q_table.get(state, np.zeros(ACTION_COUNT))
        max_future_q = np.max(q_table.get(next_state, np.zeros(ACTION_COUNT)))
        current_q = current_q_values[action]

        new_q = current_q + LEARNING_RATE * (reward + DISCOUNT_FACTOR * max_future_q - current_q)

```

```python
        # 5. Update Q-table
        if state not in q_table:
            q_table[state] = np.zeros(ACTION_COUNT)

        q_table[state][action] = new_q

        # 6. Update state
        state = next_state

    # --- End of episode ---

    # Decay epsilon
    epsilon = max(EPSILON_END, epsilon * EPSILON_DECAY)

    # Log the total reward for plotting
    episode_rewards.append(total_reward)

    # Print progress
    if (episode + 1) % 5000 == 0:
        print(f"Episode: {episode + 1}/{EPISODES} | Epsilon: {epsilon:.4f}")

env_train.close()
print("Training finished!")

# --- Step 5: Visualize Learning Curve ---

print("--- Step 5: Generating Learning Curve Plot ---")

# Calculate a moving average
chunk_size = 1000
moving_averages = [
    np.mean(episode_rewards[i:i + chunk_size])
    for i in range(0, len(episode_rewards), chunk_size)
]

plt.figure(figsize=(10, 6))
plt.plot(
    np.arange(len(moving_averages)) * chunk_size,
    moving_averages,
    label=f'Moving Average ({chunk_size} episodes)'
)
plt.title("Average Reward per Episode over Time")
plt.xlabel("Episode")
plt.ylabel("Average Reward")
plt.legend()
plt.grid(True)

# Save the plot to a file
plt.savefig("flappy_learning_curve.png")
print("Learning curve saved to 'flappy_learning_curve.png'")
# plt.show() # Uncomment this if you want to see the plot immediately

# --- Step 6: Visualize the Q-Table (Heatmap) ---

print("--- Step 6: Generating Q-Table Heatmap ---")

def get_max_q_value(state):
```

```python
    """Returns the max Q-value for a given state, or 0 if state is
    unknown."""
    return np.max(q_table.get(state, np.zeros(ACTION_COUNT)))

# Create a 2D grid representing our state space
heatmap_data = np.zeros(STATE_BINS)

for h_bin in range(STATE_BINS[0]):
    for v_bin in range(STATE_BINS[1]):
        state = (h_bin, v_bin)
        heatmap_data[v_bin, h_bin] = get_max_q_value(state) # v, h
    order for plotting

# Create the heatmap
plt.figure(figsize=(10, 8))
plt.imshow(heatmap_data, cmap='viridis', interpolation='nearest',
    aspect='auto')

plt.title("Q-Table Value Heatmap")
plt.xlabel("Horizontal Distance to Pipe (Discretized)")
plt.ylabel("Vertical Distance to Pipe (Discretized)")
plt.colorbar(label="Max Q-Value (Expected Future Reward)")

# Save the plot
plt.savefig("flappy_q_table_heatmap.png")
print("Q-table heatmap saved to 'flappy_q_table_heatmap.png'")
# plt.show() # Uncomment to show

# --- Save the Q-Table ---
print("--- Saving Q-Table to pkl file ---")
with open("flappy_q_table.pkl", "wb") as f:
    pickle.dump(q_table, f)

print(f"Q-table saved! It has {len(q_table)} states.")

# --- Step 7: See Your Trained Agent in Action! ---

print("--- Step 7: Running Trained Agent (Human View) ---")

env_human = gym.make("FlappyBird-v0", render_mode="human")

for episode in range(5):
    state_continuous, info = env_human.reset()
    state = discretize_state(state_continuous)

    done = False

    while not done:
        # We ALWAYS exploit (no epsilon)
        action = np.argmax(q_table.get(state, np.zeros(ACTION_COUNT)))

        # Take the best action
        next_state_continuous, reward, done, truncated, info =
    env_human.step(action)
        state = discretize_state(next_state_continuous)

        # Slow it down so we can watch
        time.sleep(1 / 60) # 60 FPS
```

```python

env_human.close()
print("Agent run finished.")
```

Listing 1: flappy.py: Full Training Script