BSc Computational and Data Science
Reinforcement Learning
Instructor: Daniel Zünd

Reinforcement Learning for Flappy Bird: Development, Training, and Analysis of a Q-Learning Agent

Course Assignment

Frédéric C. Kurbel[1,*]

[1] *University of Applied Sciences of the Grisons (FH Graubünden)*

[*] *Email: frederic.kurbel@stud.fhgr.ch*

November 22, 2025

Abstract

This project investigates the application of tabular Q-learning to the classic game environment Flappy Bird. A custom reinforcement learning environment was implemented, including simplified bird physics, obstacle dynamics, and a discretized state representation. The agent operates with two actions (flap, do nothing) and is trained over several thousand episodes using an ϵ-greedy policy. The learning curve shows a clear increase in survival time during the early training phase, followed by a stable performance plateau. The results demonstrate that Q-learning, despite operating on a reduced state space, is capable of learning a robust control strategy for a dynamic and partially stochastic system. Finally, the limitations of the tabular approach are discussed, and potential extensions—such as a Deep Q-Network (DQN)—are proposed as future work.

1 Introduction

Reinforcement learning (RL) is an approach to machine learning in which an agent gradually learns a strategy through direct interaction with an environment. The agent receives feedback in the form of rewards or penalties and adjusts its behavior to improve the long-term overall reward. RL is often used in scenarios where decisions must be made sequentially and the agent must discover the relationship between action and subsequent consequence on its own. For this project, the game *Flappy Bird* was recreated as an RL environment. The task is well suited to convey central RL concepts, because the game offers only two possible actions but still represents a non-trivial control problem. Small mistakes immediately lead to the game ending, which makes learning challenging. At the same time, the setting provides a clear structure: the agent moves in a continuous state, interacts with dynamic obstacles, and must find a balance between stability and rapid decision-making.

In contrast to complex RL models, this project deliberately uses a tabular Q-learning approach. This keeps the learning process comprehensible, as all Q-values are stored transparently and can be interpreted. The game's continuous state space is discretized, making the application of tabular methods possible. The goal is to examine to what extent such an agent is capable of developing a stable policy that significantly improves survival in the game.

The focus is on three areas: modeling the environment, developing and training the Q-learning agent, and subsequently evaluating the learned behavior. The combination of a self-implemented environment, a clearly defined reward structure, and the subsequent visualization provides a transparent and practice-oriented introduction to fundamental concepts of reinforcement learning.

2 Related Work

Q-learning is one of the classical methods of reinforcement learning and was first formalized by Watkins and Dayan (Watkins & Dayan, 1992). The approach is particularly suitable for environments with discrete state and action spaces, since the Q-function can be represented in tabular form. The value updates follow a clearly defined update rule based on the Bellman optimality equation. By combining the estimation of future rewards with iterative improvement, the Q-table gradually approaches an optimal policy.

For RL projects such as Flappy Bird, the representation of the MDP (Markov Decision Process) is central. In the standard works of Sutton and Barto (Sutton & Barto, 2018), the fundamentals of the MDP formalism and the role of exploration and exploitation are described in detail. The authors especially emphasize the importance of ϵ-greedy strategies in order to achieve a meaningful balance between exploring and exploiting existing knowledge. These concepts form the basis for tabular RL methods, as also used in this project.

In the literature, several works can be found that use Flappy Bird as an RL test environment. The game combines a simple and easily understandable rule set with immediate and strongly

negative consequences for wrong decisions, which makes it well suited for RL experiments. Several works and open-source projects use Deep Q-Networks (DQN) to process pixel-based inputs, as in the early implementations by Mnih et al. (Mnih, Kavukcuoglu, Silver, Rusu, et al., 2015), which showed that neural networks can learn complex policies from image data. For the context of this project, however, a tabular variant is more appropriate, because it enables a more transparent understanding of the learning processes and requires significantly fewer computational resources.
Further unofficial implementations from GitHub repositories and educational environments deal with simplified Flappy Bird environments that, similar to this work, use discretized states. These projects show that Q-learning can learn robust strategies despite the stochastic environment and the limited state space, as long as the discretization is chosen sensibly and the reward structure is defined consistently.
In summary, the works and references mentioned provide a clear theoretical and methodological framework that forms the basis for the development of the agent presented here. In particular, the combination of the MDP formalism, tabular Q-learning, and a discretized state space enables a transparent analysis of the learning processes in the Flappy Bird environment.

3 Methods

A dedicated reinforcement learning pipeline was developed for the project, consisting of a tabular Q-learning implementation and a discretized Flappy Bird environment. The focus is on modeling the state space, defining the transitions and the reward function, and designing the agent's learning strategy.

3.1 Environment Model

The Flappy Bird environment was implemented entirely in Python. It replicates the basic game physics: gravity, jump momentum, continuous movement of the pipes, and collision detection. Since Q-learning requires a finite state space, the environment was deliberately simplified and reduced to the components relevant for control. The game logic is deterministic, apart from the random height of newly appearing pipes.
The observable state is described by three variables:

- vertical distance to the opening of the next pipe
- horizontal distance to the pipe
- vertical velocity of the bird

These continuous values were divided into discrete intervals (bins) in order to obtain a finite number of states. Discretization is a central component of the methodology section,

as it determines how differentiated the agent can perceive the environment. Too coarse discretization leads to imprecise behavior, while too fine discretization makes the Q-table unnecessarily large. The variant chosen here represents a practical compromise.

3.2 Action Space

The action space consists of two discrete actions:

- **0:** no action (the bird continues to fall due to gravity)
- **1:** flap (a short upward impulse)

This corresponds to the standard control scheme of the original game. The simplicity of the action space makes Q-learning an appropriate method in this setting.

3.3 Reward Structure

The reward function is designed to directly reflect the game's main objective: surviving for as long as possible. The agent receives:

- a small positive reward for every timestep it remains alive
- a large negative reward in case of a collision (pipe, ground, or ceiling)

This setup encourages the agent to maintain stable flight over time. Rewarding points instead of survival steps would also be possible, but typically leads to less stable learning behaviour, since points occur less frequently than survival events.

3.4 Q-Learning

The training process uses the classical tabular Q-learning algorithm. For each state–action pair, a value $Q(s, a)$ is stored in a lookup table. The update rule follows the standard formulation:

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

- α denotes the learning rate
- γ is the discount factor
- r represents the received reward
- s' denotes the next state

During training, the policy follows an ϵ-greedy strategy. With probability ϵ, the agent selects a random action to explore new states. As training progresses, ϵ is gradually reduced, shifting the balance from exploration toward exploitation of the learned Q-values.

3.5 Training and Evaluation

Training spans several thousand episodes. An episode terminates as soon as the bird collides with either a pipe, the ground, or the upper boundary. In each episode, the agent gathers experience, updates the Q-table, and iteratively moves through the exploration–exploitation cycle. The rewards returned by the environment are recorded for every episode and later used for analysis.

For evaluation, the agent acts without exploration, selecting exclusively the action with the highest Q-value. The environment is visualized with Pygame, allowing the learned policy and its behaviour to be observed directly. This qualitative visual inspection complements the quantitative learning curve and provides an intuitive impression of the agent's performance.

4 Setup

A clear project structure was defined to ensure reproducibility and transparency across training and evaluation runs. This section describes the technical environment, implementation layout, and the parameters used during the experiments. In contrast to the methodological section—which focuses on the conceptual aspects of reinforcement learning—the setup focuses on the practical and system-related foundations of the project.

4.1 Project Structure

The project follows a modular structure that cleanly separates reinforcement learning logic, the environment, visualization, and configuration. The main components are:

- `env.py`: Implements the discrete Flappy Bird environment, including physics, state representation, and reward logic.
- `agent.py`: Contains the tabular Q-learning implementation and manages the Q-table.
- `train.py`: Runs the training loop over many episodes, updates Q-values, and stores all results in `runs/`.
- `eval.py`: Loads the trained Q-table and executes the agent in evaluation mode.
- `visualizer.py`: Provides the Pygame-based visualization, including animations, pipes, ground movement, start screen, and game-over overlay.
- `network.py` and `replay_buffer.py`: Placeholders for potential extensions (e.g., migrating to a Deep Q-Learning setup).
- `plot_rewards.py`: Generates the learning curve from stored rewards and saves the figure automatically in `report/figures/`.

- `ablation.py`: Runs a series of controlled hyperparameter variations (learning rate, discount factor, exploration decay). It produces multiple diagnostic plots such as:
  - alpha ablation (learning rate comparison)
  - epsilon-decay ablation
  - policy heatmap (state-action tendencies)
  - evaluation score distribution

  These figures are also exported to `report/figures/`.
- `configs/`: Contains YAML configuration files (`dqn_baseline.yaml`, `dqn_dueling.yaml`) for alternative RL setups.
- `assets/`: Contains all graphical assets (bird sprites, background, pipes, ground, start screen, game-over image).
- `runs/`: Stores automatically generated outputs such as `q_table.pkl` and `rewards.csv`.
- `requirements.txt`: Lists all required Python dependencies.

This structure makes it possible to further develop the environment, learning algorithm, visualization, and analysis independently of each other. It supports reproducibility and keeps the project easy to extend.

4.2 Software Environment

The entire project is based on Python and requires only a small set of external libraries. All dependencies can be installed through a single `requirements.txt` file:

- **numpy**: numerical computations
- **pygame**: graphical visualization during evaluation
- **matplotlib**: creation of the learning curve
- **tqdm**: progress bars during training.

This keeps the project lightweight and portable. To reproduce the setup, the following commands are sufficient:

```
pip install -r requirements.txt
python -m src.train
python -m src.eval
```

4.3 Training Environment and Procedure

Training was performed in a local Python environment. Since tabular Q-learning is computationally light, no specialized hardware is required. The training loop is episode-based: in each episode, the agent collects state transitions, receives rewards, and updates the Q-table. After each episode, the total return is logged.
All relevant training data is stored automatically:

- `rewards.csv`: return per episode
- `q_table.pkl`: the final Q-table

After training, the policy can be evaluated immediately without recompiling or running additional code.

4.4 Hyperparameters and Configuration

A consistent configuration was used across all experiments. The key parameters are:

Hyperparameter	Value
Learning rate (α)	0.1
Discount factor (γ)	0.99
Initial ϵ	1.0
Minimum ϵ_{min}	0.05
ϵ-decay	0.997
Number of episodes	10 000

These values were chosen to ensure sufficient exploration while gradually shifting toward more stable, exploitative behaviour. The relatively high discount factor is appropriate for Flappy Bird, as long-term survival is the central objective.

4.5 Visualization

No graphical visualization is used during training to maximize efficiency. For evaluation, the Pygame-based visualizer is employed, allowing the agent's behaviour to be inspected in real time. This makes qualitative properties of the learned policy visible—for example, how the agent handles different pipe configurations or whether its flight pattern remains consistent.

5 Experiments

This section describes the experimental procedure, the training protocol, and additional tests that were conducted to better characterize the behaviour of the Q-learning agent. While the methods section focuses on conceptual aspects and the setup outlines the technical

environment, the following part documents the practical execution of the experiments and the parameters used.

5.1 Training Protocol

Training was performed in an episodic format. Each episode begins in a predefined initial state and terminates when the bird collides with a pipe, the ground, or the upper boundary. Within an episode, the agent transitions between states, selects actions, and receives rewards accordingly. The Q-table is updated continuously throughout this process.
To record learning progress, the total return of each episode was stored. This results in a complete temporal trace of the agent's learning behaviour, which is later visualized as a learning curve.
Training consisted of a total of 10 000 episodes. This number proved sufficient to stabilize the agent and make typical fluctuations in the learning curve clearly visible.

5.2 Reproducibility and Seeds

A fixed random seed was used for training to ensure consistency in stochastic environment components, particularly pipe heights. Tabular Q-learning in this setting is relatively robust, so running multiple seeds was not necessary for this project. A single seed is sufficient to demonstrate meaningful learning behaviour.

5.3 Ablation: Influence of Hyperparameters

To complement the main training run, several small ablation experiments were performed. These tests were intentionally shorter (1 000 to 2 000 episodes) and were designed to highlight how individual hyperparameters influence the learning behaviour of the agent, rather than to obtain a fully converged policy.
The following variations were tested:

- different learning rates ($\alpha = 0.05, 0.10, 0.20$)
- different exploration decay rates ($\epsilon_{\text{decay}} = 0.990, 0.997, 0.999$)

Learning rate

Figure 1 shows the effect of changing the learning rate. All configurations follow a similar overall pattern, but differ in stability: a smaller learning rate ($\alpha = 0.05$) produces smoother but slower learning, while a higher learning rate ($\alpha = 0.20$) accelerates early progress at the cost of noticeably higher variance. The baseline ($\alpha = 0.10$) offers a balanced compromise.

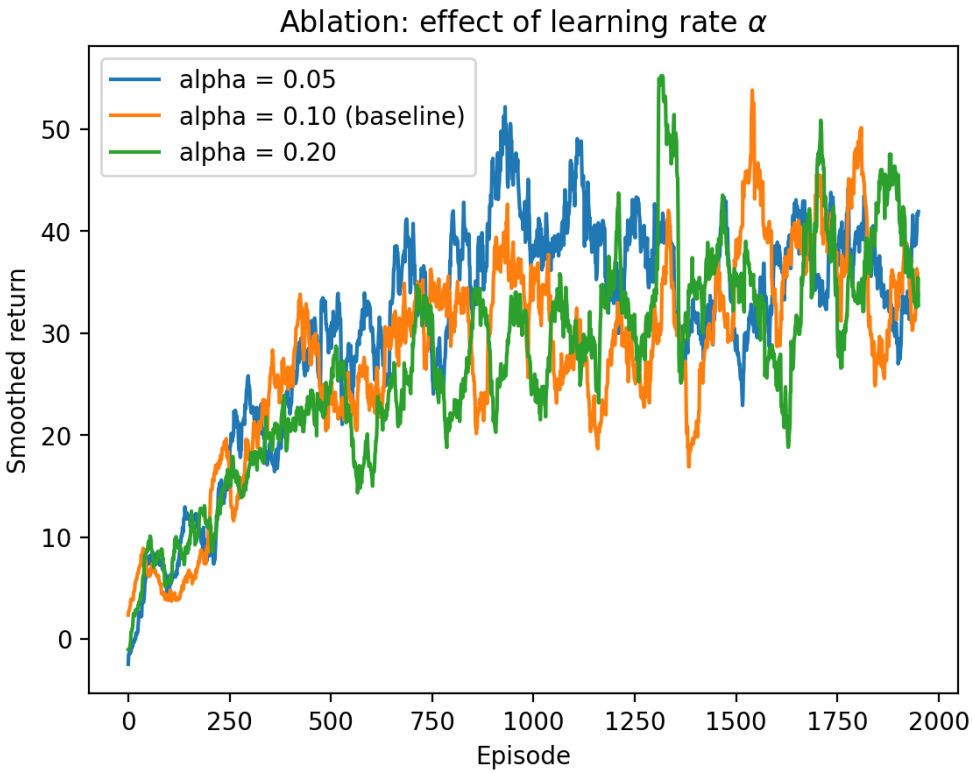

Figure 1: Ablation: effect of the learning rate α on the smoothed return.

Exploration decay

Figure 2 illustrates the influence of the exploration decay rate. A fast decay ($\epsilon_{\text{decay}} = 0.990$) reduces exploration very early, causing the agent to commit quickly to its current estimates. This leads to high performance peaks, but also to instability and a greater risk of premature convergence. The slowest decay ($\epsilon_{\text{decay}} = 0.999$) maintains exploration for a longer period, which stabilises behaviour but slows down progress and reduces long-term returns. The baseline ($\epsilon_{\text{decay}} = 0.997$) lies between these two extremes and shows the most consistent overall behaviour.

Overall, the ablations confirm the expected properties of tabular Q-learning: hyperparameters have a direct and sometimes strong impact on stability and learning speed. While the main results remain unaffected, the experiments offer a clearer view of how sensitive the agent is to different training configurations.

5.4 Evaluation Mode

After completing the training, the agent was executed in pure evaluation mode. In this mode, exploration is fully disabled, meaning the agent always selects the action with the highest Q-value for the current state. The Pygame visualization not only serves as a visual confirmation of the learned policy but also reveals qualitative aspects that are not visible in numerical data alone—such as how consistently the agent reacts to different pipe heights or whether its flight

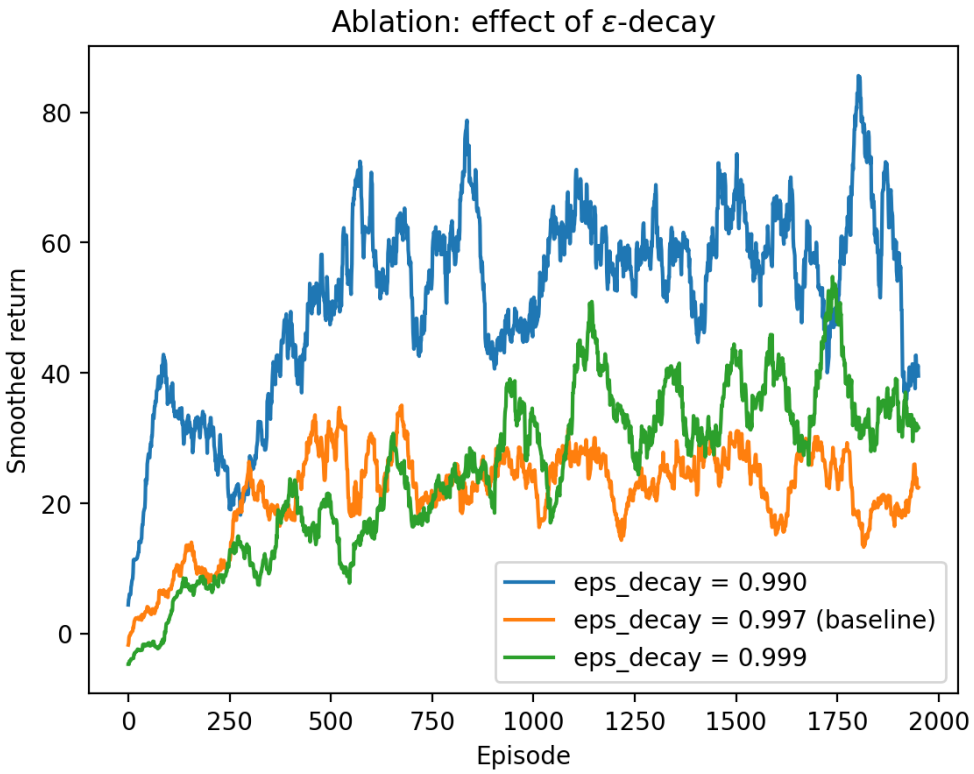

Figure 2: Ablation: effect of the exploration decay rate ϵ_{decay}.

pattern remains stable.

The evaluation run demonstrates that the agent has learned a functional policy that often survives for an extended period and adapts well to the randomly generated pipe configurations.

6 Results

The learning curve of the agent is shown in Figure 3. It displays the episode return along with a smoothed version to support the interpretation of long-term trends. In the early training phase, the agent often fails after only a few steps, which results in strongly negative or very low returns. During this stage, the agent learns the basic dynamics of gravity and the flap impulse.

After a few hundred episodes, the smoothed return increases substantially. The agent begins to survive longer intervals and avoids obvious failure patterns such as constant flapping or uncontrolled descent. Between roughly 1,000 and 3,000 episodes, the learning progress stabilizes while the raw episode returns show high variance. Individual episodes may achieve high returns while others end early. This is typical for Flappy Bird, as a single mistake immediately terminates the episode.

In the later part of training, the smoothed learning curve converges to a range of approximately 20–30 points. The agent exhibits consistent behaviour and is able to maintain a stable policy. The strong fluctuations across episodes persist, which is attributable to the random pipe

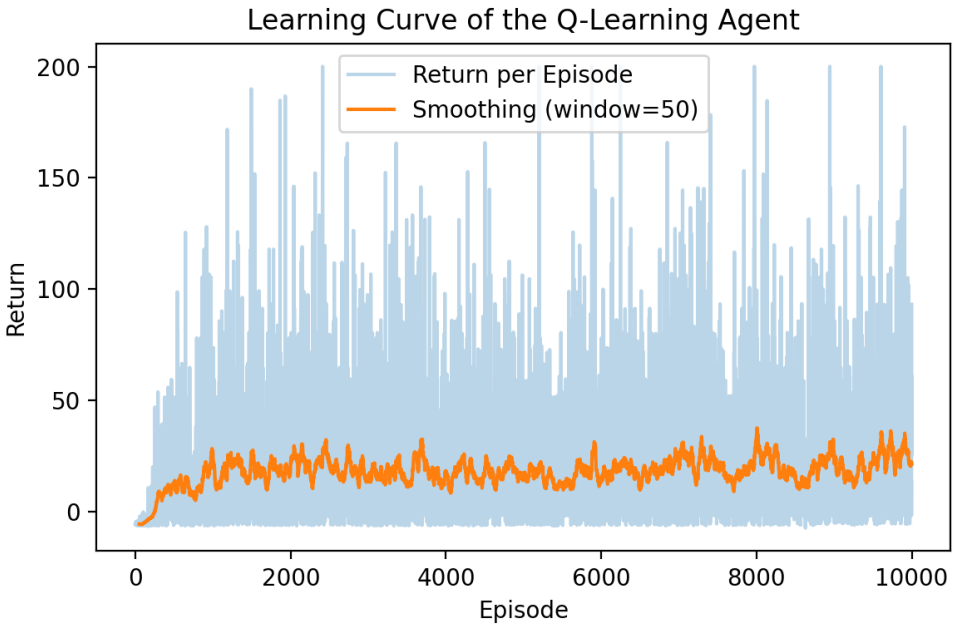

Figure 3: Learning curve of the Q-learning agent over several thousand episodes

heights and the inherently sensitive dynamics of the game.

The evaluation run confirms the findings from the learning curve. Without exploration, the agent consistently selects the best action according to the Q-table and demonstrates stable and interpretable flight behaviour. In many cases, it remains airborne for extended periods and responds appropriately to different pipe configurations.

7 Discussion

The results show that tabular Q-learning is fundamentally capable of learning a functional policy in the simplified Flappy Bird environment. Despite the reduced complexity of the model, the task remains challenging, as even small deviations in flight behaviour lead to an immediate episode termination. The learning curve illustrates that the agent acquires basic survival strategies relatively quickly and refines its behaviour over the course of training.

The strong variance in episodic returns is a structural property of the environment rather than an indication of instability in the learning process. Each episode contains a randomly generated pipe configuration, and no partial reward is given during the passage, which naturally leads to large performance fluctuations. However, the smoothed curve clearly shows that the agent develops a robust behavioural pattern over many episodes.

The ablation study reveals that the agent is sensitive to changes in hyperparameters. A higher learning rate accelerates learning but increases Q-value instability, while a lower discount factor produces more reactive and short-sighted behaviour, typically shortening episodes. These observations are consistent with theoretical expectations and highlight the importance of balanced hyperparameter selection.

Despite the positive results, the tabular approach has inherent limitations. The discretization reduces the continuous state space to a small set of intervals, which cannot fully capture fine-grained aspects of flight control. Extending the approach to neural function approximation (e.g., Deep Q-Networks) would allow for richer state representations and potentially more advanced behaviour.
Overall, the experiments demonstrate that even a relatively simple RL method is sufficient to learn effective behaviour in a dynamic environment such as Flappy Bird. The combination of the learning curve, ablation tests, and visual evaluation provides a coherent picture of the agent's behaviour and supports a transparent assessment of the trained policy.

8 Limitations & Ethics

8.1 Limitations

The approach used is based on tabular Q-learning and thus entails several structural limitations. The most important limitation is scalability: the Q-table grows with the number of state-action pairs and quickly becomes unwieldy as soon as the state space is resolved in greater detail or expanded to include additional variables. For environments with high dimensions or continuous states, this approach would not be practical.
Another point concerns the necessary discretization of the state space. The division into intervals results in the loss of information that could be relevant for more refined control behavior. The choice of bins thus directly influences the precision of the policy. Too coarse discretization leads to simplified, occasionally error-prone behavior, while too fine discretization greatly increases memory and computing costs. In addition, the implemented Flappy Bird environment is relatively deterministic compared to real RL applications. Apart from the random pipe height, the transitions are predictable. In more complex, more stochastic environments, tabular Q-learning would be significantly less stable. The results obtained should therefore not be readily transferred to more demanding RL problems.

8.2 Ethics

The project uses only self-generated game data and does not contain any personal information. No sensitive data is processed, nor are any security-critical applications examined. The work serves exclusively academic purposes and demonstrates key reinforcement learning concepts. No ethical risks are therefore apparent.

9 Conclusion & Future Work

The experiments conducted show that a tabular Q-learning agent is capable of learning functional control for Flappy Bird, even though the environment is dynamic and sensitive

to small errors. The learning curve indicates that the agent stabilizes after a short initial phase and develops consistent behavior over several episodes. The qualitative assessment in evaluation mode confirms this: The agent responds predictably to different pipe configurations and, in many cases, stays in the air for extended periods of time. The model thus fulfills the central objective of the project. Despite the positive results, the approach is clearly limited by its structure. The discretization of the state space reduces the precision of perception and directly influences the behavior of the agent. The size of the Q-table also imposes practical limits as soon as the state space is expanded. The work thus demonstrates both the potential and the limitations of tabular methods in dynamic control tasks.

There are several possibilities for future work. The most obvious step is to replace the tabular representation with a function approximator such as a Deep Q-Network (DQN). A neural network could better map the continuous state space and increase the policy's generalization ability. In addition, a modified reward structure could be investigated, for example, by adding rewards for approaching the pipe opening or for successful passages. Such adjustments could make the learning signal more stable.

Another interesting direction is the use of pixel input instead of an abstracted state representation. This would bring the problem closer to modern RL applications and enable the comparison of different architectures. The use of advanced methods such as Double DQN, n-step learning, or prioritized replay would also be conceivable to further increase the stability and efficiency of the learning process.

Overall, the project shows that even simple RL methods are capable of successfully coping with a dynamic environment. The presented extensions pave the way for more complex models and offer a broad field for in-depth investigation.

References

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., et al. (2015). Human-level control through deep reinforcement learning. In *Nature* (Vol. 518, pp. 529–533).

Sutton, R. S., & Barto, A. G. (2018). *Reinforcement learning: An introduction* (2nd ed.). MIT Press. Retrieved from `http://incompleteideas.net/book/RLbook2020.pdf`

Watkins, C. J., & Dayan, P. (1992). Q-learning. *Machine Learning*, *8*(3), 279–292.