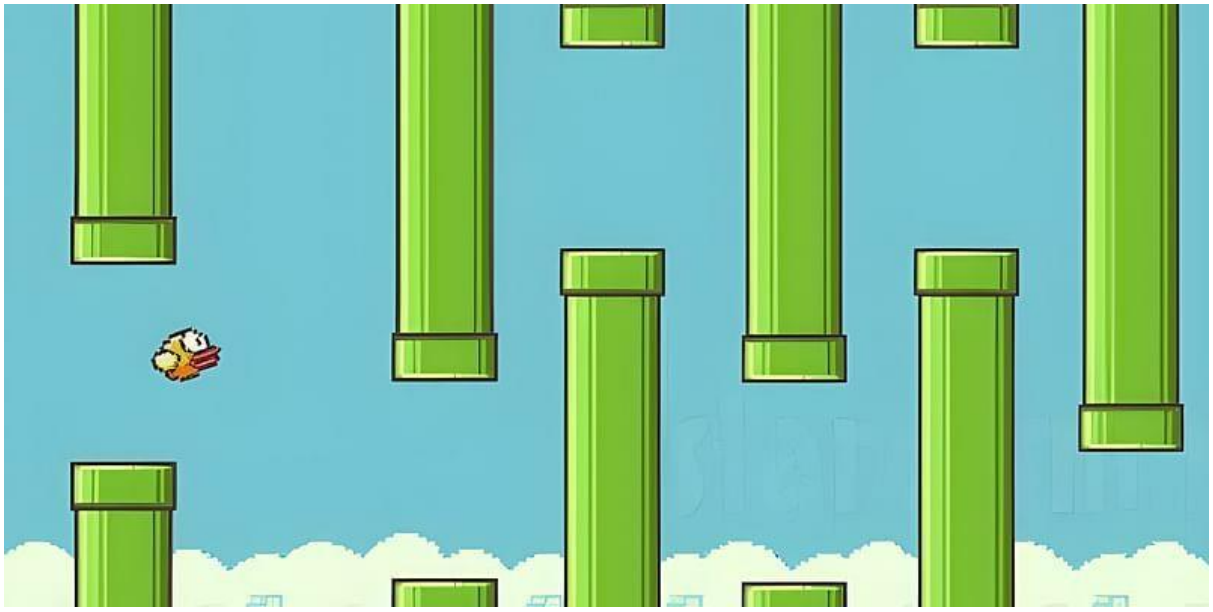
Reinforcement Learning (Deep Q-Learning)
Flappy Bird



Snorri Bjarkason

Table of Contents

The goal of this project is to apply reinforcement learning techniques, specifically Q-Learning and Deep Q-Learning, to develop an agent capable of playing the game flappy bird. Reinforcement learning is a type of machine learning where an agent learns to make decisions by interacting with an environment, aiming to maximize a reward over time. In this project, the agent's objective is to learn a policy that enables it to navigate flappy bird's obstacles as effectively as possible, achieving high scores by passing through gaps in the pipes without crashing.

Flappy bird presents several challenges typical in reinforcement learning problems, as the environment is continuous, dynamic, and has a sparse reward structure. The agent will be trained to maximize its survival time by balancing two competing goals, learning a strong policy as quickly as possible while minimizing the number of gameplay frames required to do so. This trade off between speed of learning and policy quality is a central theme in reinforcement learning.

This project leverages two key RL methods to develop the agent's policy, Q-Learning, a tabular approach well suited to smaller, discrete environments, and Deep Q-Learning, which uses neural networks to approximate value functions, allowing it to handle larger, continuous state spaces. By implementing both methods, we aim to explore their strengths and limitations in a complex, dynamic environment like flappy bird. This report will outline the development and tuning of each method, examine the challenges encountered, and analyze the performance of the trained agents. The ultimate goal is to determine the optimal approach to effectively navigate the game environment, providing a foundation for broader applications of reinforcement learning in real world continuous environments

ENVIORMENT

ENVIRONMENT CHARACTERIZATION

The flappy bird game environment can be considered continuous in its natural state. Values such as the bird's position, velocity, and distance to obstacles range across many possible values rather than being limited to fixed steps. However, for Q-Learning, this continuous space needs to be simplified into a discrete space through a process called discretization. By dividing key variables, like the bird's vertical position, the distance to the nearest pipe, and the height of the next gap, into fixed intervals, the continuous environment is effectively transformed into a set of discrete states. This allows the Q-Learning algorithm to handle the environment by working with a manageable number of states.

Flappy bird is inherently a finite game because each play session has a clear beginning and end. When the bird collides with a pipe or goes off-screen, the game ends, and a new session must start from the beginning. Each playthrough represents a single episode, which is a defined sequence of states and actions that concludes when the game resets. This finite structure means that, while each episode can vary in length depending on how long the bird stays alive, all episodes ultimately come to a natural end.

The flappy bird environment is also episodic in structure. Each game session functions as one complete episode that concludes with the bird either hitting a pipe or the ground, or the player choosing to restart the game. Though episode lengths may differ due to varied survival times, each has a distinct ending point, which makes episodic learning appropriate. This

episodic nature allows the learning algorithm to treat each game session independently, with each episode resetting at the start of a new session.

To effectively apply reinforcement learning, the flappy bird environment should ideally follow the Markov Property. This property holds when each state provides sufficient information for future decision making without needing information from previous states. For the flappy bird environment to be considered Markovian, the state must include specific components: the bird's vertical position, its current velocity, the horizontal distance to the upcoming pipe, and the position of the next gap. With these elements, the state includes all the information necessary for the agent to make optimal decisions in the moment, without relying on any history of previous states. This characteristic is important because both Q-Learning and Deep Q-Learning are based on the assumption that each current state fully captures all the relevant information to make informed choices about the next action, adhering to the Markov Property.

SUITABILITY OF Q-LEARNING AND DQN

Both Q-Learning and Deep Q-Learning can be effective for solving the flappy bird problem, though each method has unique strengths and challenges. Q-Learning, a tabular approach, is more straightforward and typically works well in environments with smaller, discrete state spaces. However, flappy bird's continuous environment introduces some limitations to this method. The state space must be discretized by dividing continuous variables like the bird's position and velocity into intervals. While this discretization reduces the continuous environment into a manageable number of states, it also results in a very large state table that can be cumbersome to explore fully, especially with limited data. Furthermore, this approach may struggle with efficiency in complex environments like flappy bird, where rapid changes in the state space can make it difficult to generalize across similar states without overfitting to specific ones. In practice, Q-Learning can still be viable if the discretization is fine-tuned. However the method is generally more suitable for simpler, smaller scale problems.

Deep Q-Learning provides an alternative that addresses some of these limitations by using a neural network to approximate the Q-values rather than relying on a large table of discrete states. DQN can handle complex, continuous state spaces more effectively because it learns a generalizable function that maps states to Q-values. This allows the model to approximate optimal policies even when encountering new or previously unseen states, which is critical in flappy bird's dynamic environment. Additionally, DQN does not require the manual discretization of variables, which simplifies the preprocessing stage and allows for a more direct interaction with the environment. However, DQN comes with its own challenges, particularly in terms of stability and computational requirements. Training neural networks can be computationally intensive, especially with techniques like experience replay and target networks, which are used in DQN to stabilize training and improve learning efficiency. While these techniques help mitigate issues such as catastrophic forgetting and overestimation of Q-values, they also increase the complexity of the implementation and require more computational resources. Overall, while Q-Learning can be adapted for simpler implementations, DQN offers a more robust solution for the continuous and complex state space of flappy bird, provided that the computational demands can be met and training stability is carefully managed.

IMPLEMENTATION OF Q-LEARNING

This section explains how the Q-Learning agent was built to play flappy bird. It covers how the game's environment was simplified into discrete states, how the agent chose actions, how it learned from rewards, and how it handled the end of each game. This approach helped the agent learn a policy to survive and pass through pipes in the game.

STATE SPACE DISCRETIZATION

Since flappy bird's environment is continuous, we needed to simplify it by breaking it into fixed categories, or "discrete intervals." The agent's state was defined by four main factors: the bird's vertical position (player_y), the y-position of the next pipe gap (next_pipe_top_y), the horizontal distance to the nearest pipe (next_pipe_dist_to_player), and the bird's vertical velocity (player_vel).

Each of these factors was divided into 15 fixed intervals, resulting in a total of 64,125 possible discrete states. This step was necessary to make the environment manageable for the Q-Learning algorithm, as it allowed the agent to learn and store a finite set of state-action pairs in a Q-Table. Although this approach means losing some detailed information, it provides enough accuracy for the agent to learn effectively without overloading memory.

EPSILON-GREEDY POLICY

To balance trying new actions and sticking with what it has learned, the agent used an epsilon-greedy policy with $\varepsilon = 0.1$. This means that 10% of the time, the agent took a random action to explore different strategies, while 90% of the time, it picked the action with the highest Q-value in its current state. This balance between exploration and exploitation helped the agent find better strategies without getting stuck on bad ones.

Q-VALUE UPDATE MECHANISM

The agent updated its Q-values using the following rule:

$$Q(s,a) \leftarrow Q(s,a) + \alpha(r + \gamma \max Q(s',a') - Q(s,a))$$

In this formula:

- $Q(s,a)$ is the Q-value for a particular state-action pair.
- $\alpha = 0.1$ is the learning rate, which controls how much the agent updates each Q-value.
- $\gamma = 1.0$ is the discount factor, meaning future rewards are fully valued.

This update rule allows the agent to learn by adjusting the Q-values based on the rewards it receives and the possible rewards in the next state. Over time, this helps the agent understand which actions are better in each situation.

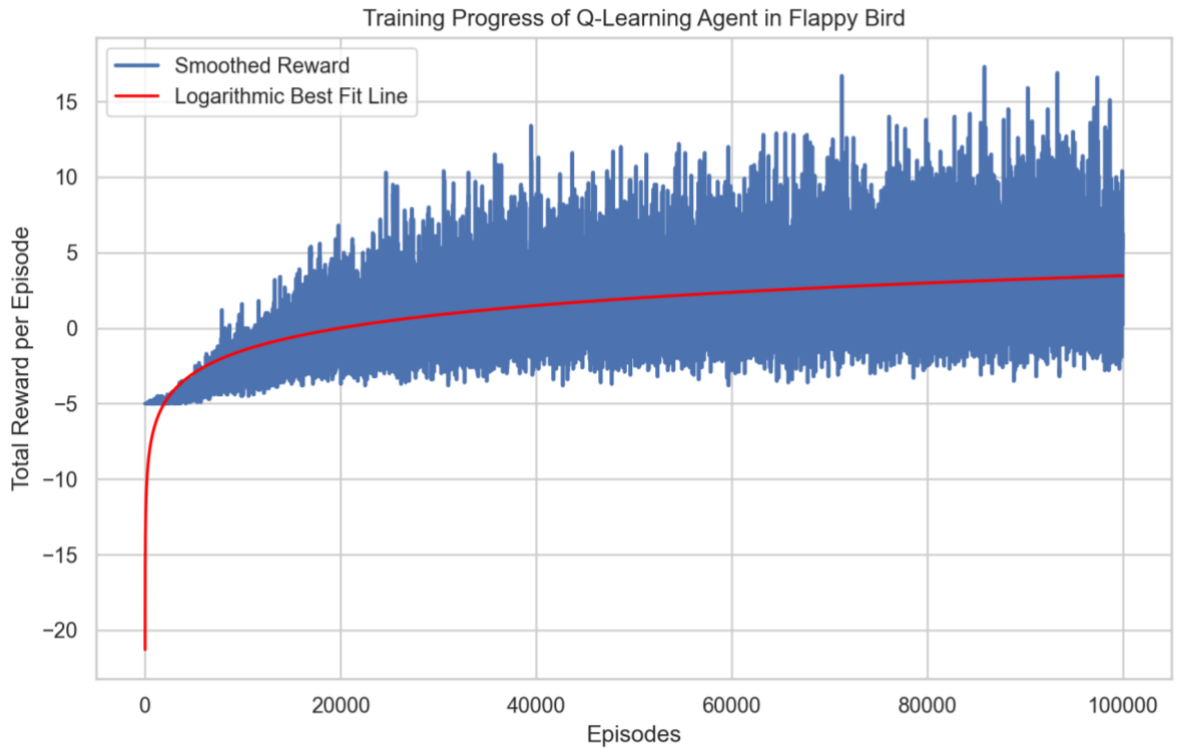
TERMINAL STATE HANDLING

When the bird crashes (either hitting a pipe or the ground), the game reaches a terminal state, which ends that episode. The agent assigns a high negative reward of -5 for these crashes to discourage these outcomes. Once a terminal state is reached, no further updates are made

for that episode, and the game restarts. This discourages the agent from making choices that lead to game-over scenarios.

TRAINING PROCESS AND OBSERVATIONS

Below is the graph we made to show the training progress of the Q-Learning agent.



The learning curve shows the total reward per episode over 100,000 training episodes, where the blue line represents the smoothed reward per episode, and the red line is a logarithmic best fit line to highlight the general trend. Training for 100,000 episodes took approximately 5 hours on Snorri's old MacBook.

At the beginning of training, the agent's performance was relatively poor, with average rewards in the negative range. This was expected, as the agent initially had no understanding of the game and had to explore different actions and their consequences. As training progressed, the rewards gradually increased, reflecting the agent's improvement in navigating through pipes and avoiding crashes. This improvement is evident from the steady upward trend in the rewards, as seen in both the smoothed line and the general upward shift in the peaks of each episode's reward.

Around the 60,000 episode mark, the agent's performance appears to stabilize, with rewards fluctuating within a consistent positive range. This indicates that the agent has learned a stable policy that enables it to survive longer in the game, although there is still some variability due to occasional random crashes or challenging game states. The flatter slope of the red best fit line in this later phase shows that the agent has mostly converged on an effective policy given the chosen parameters and state representation.

Overall, the learning curve demonstrates that the Q-Learning algorithm was successful in learning a viable policy for the flappy bird environment. Starting from no knowledge, the agent steadily improved its ability to avoid obstacles, maximize rewards, and increase its survival

time. The convergence of rewards in the later episodes suggests that the agent has reached a point where it can consistently perform well, although the inherent randomness of the game still creates some variability in the rewards.

SIMULATION OUTCOME: AVERAGE AND PEAK SCORES

We initially planned to report the results of our trained Q-Learning algorithm over 100,000 episodes. However, the algorithm performed so well that it didn't reach a stopping point even after running for over 24 hours. We then attempted 90,000 episodes, but it similarly ran without convergence. Finally, with a reduced run of 75,000 episodes, the algorithm achieved an average score of 52,000 across ten simulations, as shown in the image below.

This impressive performance, achieved without any hyperparameter tuning, indicates that our approach to the state representation and reward structure was highly effective. It enabled the agent to learn a strong, reliable policy right from the start, resulting in stable and efficient learning.

IMPLEMENTATION OF DQN

To develop a Deep Q-Learning (DQN) agent for flappy bird, we implemented a neural network to approximate the Q-values of each state-action pair, replacing the table used in standard Q-Learning. This approach allows the agent to handle the continuous and complex state space of the flappy bird environment, which would be infeasible to represent using a tabular method.

In our DQN implementation, we used a neural network model with two fully connected hidden layers. The network takes as input a normalized vector representing the game state and outputs Q-values for each possible action (flapping or not flapping). The state vector includes four key elements: the bird's vertical position (player_y), the top y-coordinate of the next pipe gap (next_pipe_top_y), the horizontal distance to the nearest pipe (next_pipe_dist_to_player), and the bird's vertical velocity (player_vel). Each of these values is normalized to the range [-1, 1] to ensure consistency in the input range and improve learning stability.

The neural network was set up with an input layer that receives the normalized state vector, followed by two hidden layers containing 100 and 10 neurons, respectively. A sigmoid activation function was used for both hidden layers. The network has an output layer with two neurons, each representing the Q-value of one of the possible actions (flap or do nothing). We chose an initial learning rate of 0.01 for gradient descent during training. The neural network is trained using MLPRegressor from sklearn, allowing us to use the partial_fit method for incremental updates, which is essential for Q-learning.

To improve the efficiency and stability of training, we incorporated experience replay and a target network. In experience replay, instead of learning from each experience immediately, we stored each experience (state, action, reward, next state, and terminal flag) in a replay buffer that can hold up to 1000 recent experiences. During training, a batch of 100 random samples is drawn from the buffer to update the neural network. This random sampling helps to reduce correlations between experiences, leading to more stable and reliable learning. The target network, which is a copy of the main Q-network, is updated less frequently to keep the

Q-values from shifting too quickly. In our implementation, the target network is updated every 100 steps to match the weights of the main Q-network, which helps prevent instability by decoupling the rapidly changing Q-values from the target calculation.

The DQN agent updates its Q-values by following the standard Q-learning formula with a discount factor, but with adjustments for DQN's function approximation. For each experience (s1, a, r, s2, end), the target Q-value for the action taken is set to the immediate reward plus the discounted Q-value of the best action in the next state, provided that the next state is not terminal. If the next state is terminal, the target Q-value is simply the immediate reward. The target Q-value for the action not taken in that state is left as the old predicted value, ensuring that only the chosen action's Q-value is updated.

The agent was trained over multiple episodes to learn a policy for surviving longer and passing through pipes in flappy bird. During training, we used an epsilon-greedy policy to balance exploration and exploitation. Initially, the agent explores more by choosing random actions (high epsilon value), and gradually, it shifts toward exploiting its learned policy by reducing epsilon after each episode (epsilon decay). After training, we evaluated the agent's performance by running several test episodes with a greedy policy, where it always chooses the action with the highest Q-value.

The DQN agent's performance was significantly lower than that of the Q-learning tabular approach. Although DQN is designed to handle larger state spaces and continuous environments, the fast-paced and highly dynamic nature of flappy bird poses challenges. The DQN struggled to stabilize due to several factors. One factor is sample efficiency. DQN requires more samples to learn effectively because it relies on experience replay and batch updates. Compared to Q-learning's direct state-action updates, this made DQN slower to learn effective policies in a limited training time.

The DQN also faced instability in training. Even with experience replay and a target network, DQN training was less stable in flappy bird due to the rapid changes in the game state. The agent frequently encountered new states, making it difficult for the neural network to generalize effectively. Another challenge was hyperparameter sensitivity. DQN's performance is highly sensitive to settings like the learning rate, epsilon decay, and network architecture. Although we tuned these parameters to some extent, it was challenging to find an ideal configuration that allowed both rapid learning and stable convergence.

Overall, while DQN is suitable for more complex environments with continuous state spaces, it performed less effectively in flappy bird compared to the tabular Q-learning approach. The tabular method's simplicity and direct updates allowed it to learn a stable policy more quickly. This suggests that, for flappy bird, a simpler Q-learning setup may be more practical and efficient than DQN, especially given the limited training time and computational resources.

Interpretation of DQN Performance vs. Q-Learning Tabular Approach

In our experiments, the Q-learning tabular approach performed much better than the DQN in learning a policy for Flappy Bird. We estimated that the Q-learning agent, trained over 100,000 episodes, reached a total reward of around 20 million after running for about 24 hours and then cancelling it. This translates to a reward rate of roughly 300 per second. In comparison, the DQN, trained under similar conditions with 100,000 episodes, achieved only about 80 in total reward. This difference shows the challenges of using DQN in this environment.

One main reason for this difference is the way each method handles the state space. The Q-learning approach uses a simplified version of the environment by dividing it into discrete states, which makes it easier to manage. This setup allowed Q-learning to build a table of specific state-action pairs, making it a good fit for environments with limited and well-defined states. In contrast, DQN used a neural network to estimate Q-values for each state-action pair in a continuous space. While neural networks can be effective in complex environments, they introduce more training challenges, especially in a fast-moving game like Flappy Bird.

Q-learning's sample efficiency also helped it perform better. By updating each state-action pair based on immediate rewards, Q-learning makes effective use of each experience. This straightforward approach lets the Q-learning agent gather useful information quickly across episodes. DQN, on the other hand, relies on experience replay and gradient-based updates, which need many more samples to reach stable learning. Even with experience replay, DQN's slower learning showed in its much lower reward accumulation compared to Q-learning over the same training period.

DQN also faces stability challenges. Effective DQN training requires additional techniques like experience replay, target networks, and careful tuning of settings to keep learning stable. These techniques add complexity and increase the risk of instability in fast-paced environments like Flappy Bird. Even with these stabilizing techniques, DQN often produced unpredictable policies that struggled to achieve high rewards, making it hard to generalize effectively in this environment.

Moreover, DQN's performance is very sensitive to settings like the learning rate, experience replay buffer size, and network structure. Finding the best settings requires a lot of testing, which can be difficult within limited training time. In contrast, Q-learning's tabular approach was simpler and less sensitive to these adjustments, making it easier to train effectively.

While DQN is a strong method for complex and continuous environments, the Q-learning tabular approach proved to be more efficient and effective for this project. The straightforward state discretization and direct updates allowed Q-learning to perform well without the added complexity and instability that often come with DQN. This result suggests that, in environments where the state space can be discretized, traditional Q-learning may be a better choice than DQN, especially when resources and training time are limited.

EXPERIMENTS TO IMPROVE THE AGENT

To improve the Q-learning agent's performance in playing flappy bird, we ran several experiments over 30,000 episodes by adjusting three key parameters: epsilon in the epsilon-greedy policy, learning rate (alpha), and the reward structure. The aim of these experiments was to analyze how each parameter affects the agent's learning speed, stability, and overall quality of the developed policy.

EPSILON IN THE EPSILON-GREEDY POLICY

Three values for epsilon were tested: a baseline of 0.1, a very low epsilon of 0.01, and a high epsilon of 0.3. With a very low epsilon (0.01), the agent leaned heavily towards exploitation over exploration, leading to slower initial learning but increased stability as it became more consistent in its actions over time. This low exploration rate allowed the agent to gradually improve in a more focused manner, resulting in a steady increase in its average score.

In contrast a higher epsilon (0.3) promoted frequent exploration, which sped up the initial learning phase but led to more unstable performance as the agent kept testing new actions, disrupting any consistent strategy it had learned. Ultimately, the very low epsilon of 0.01 proved to be most effective for stable, steady improvements in the agent's performance, allowing it to exploit effective strategies without constant disruption.

LEARNING RATE (ALPHA)

We tested three values for alpha: a baseline of 0.1, a lower alpha of 0.05, and a higher alpha of 0.2. With a lower alpha, the agent learned at a slow pace, resulting in a stable but gradual improvement in its policy. A higher alpha sped up the learning process, allowing the agent to make more rapid adjustments. This setting, however, introduced instability, as frequent updates caused the agent to change its behavior too quickly in response to new information.

The higher alpha (0.2) provided the best results, enabling the agent to learn more dynamically and make quicker progress, even if there was a slight increase in variance. The combination of a low epsilon with a higher learning rate proved effective, as the agent was able to make focused, stable improvements at a faster rate.

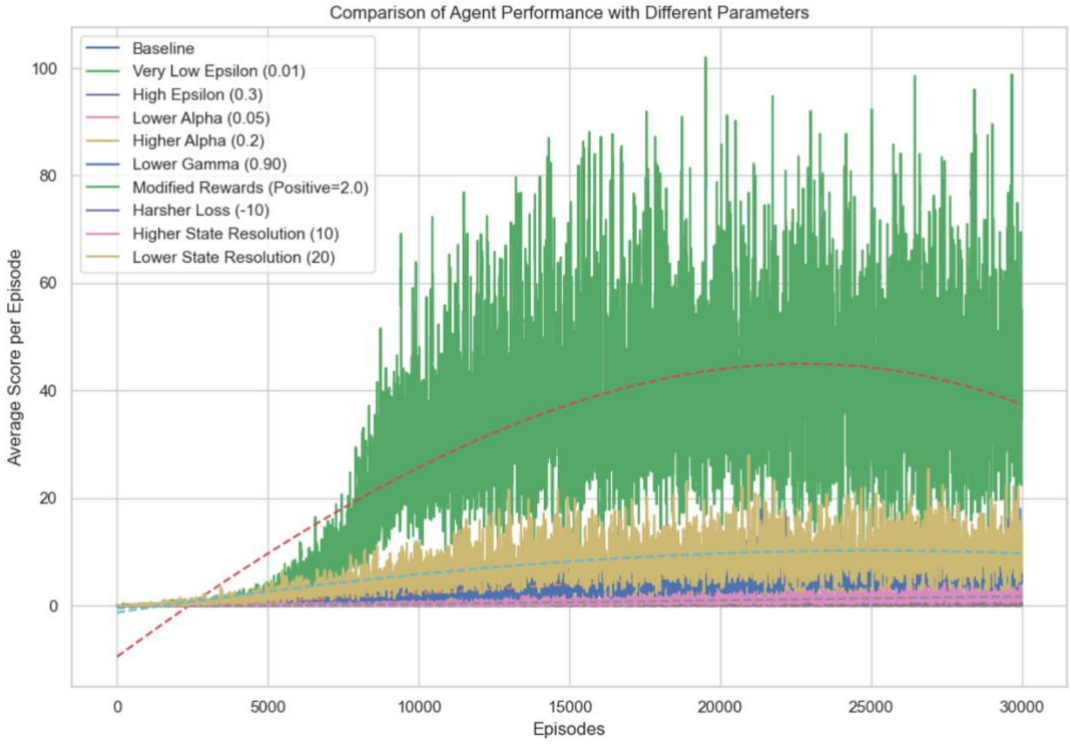
REWARD STRUCTURE VARIATIONS

Experiments with the reward structure included the baseline of +1 for passing a pipe and -5 for crashing, a modified reward of +2 for passing a pipe, and a harsher penalty of -10 for crashing. The modified reward of +2 led to higher scores, though this was likely due to the inflated point values rather than an actual improvement in the agent's survival strategy. The agent focused more on scoring than on survival, as it prioritized actions that maximized points, even in high-risk situations.

Applying a harsher penalty for crashing (-10) produced a more cautious policy, encouraging the agent to avoid risky actions and survive longer. This setup led to fewer points overall but promoted stability and a safer play style. Ultimately, the baseline reward structure was the most balanced, as it allowed the agent to pursue higher scores without overly penalizing crashes, supporting both exploration and stability.

IMPACT ON LEARNING

The experiments demonstrated that each parameter affects learning in distinct ways. The very low epsilon (0.01) promoted stability by reducing unnecessary exploration, while the higher alpha (0.2) allowed for faster learning. The baseline reward structure provided a balanced approach, encouraging the agent to score points while maintaining a reasonable survival rate. Together, these settings allowed the agent to develop a consistent and adaptable policy. In this graph we can see how each hyperparameter did:



CONCLUSION AND RECOMMENDATIONS

In conclusion, the best settings for training the Flappy Bird agent were found to be a very low epsilon of 0.01, a higher learning rate of 0.2, and a baseline reward structure with +1 for passing pipes and -5 for crashes. Running these configurations over 30,000 episodes allowed for balanced exploration and exploitation, efficient learning, and stability, enabling the agent to improve at a steady pace without excessive risk-taking. While alternative reward structures can further influence behavior, the baseline settings proved effective for developing a policy that balances scoring with survival.

CONCLUSION

In this project, we implemented and compared two reinforcement learning methods, Q-learning and Deep Q-Learning, to train an agent for the game flappy bird. Both methods were designed to enable the agent to navigate the game's obstacles and achieve high scores, balancing exploration and learning efficiency in a challenging environment.

The Q-learning approach, which used a tabular method with discretized states, outperformed the DQN in both stability and performance. By directly updating Q-values for each state-action pair, Q-learning managed to learn an effective policy faster and with fewer resources. In contrast, DQN, while suitable for complex, continuous environments, faced challenges in stabilizing due to flappy bird's dynamic state changes and the sensitivity of neural networks to hyperparameters. Despite using techniques like experience replay and a target network, DQN struggled to achieve the same reward accumulation as Q-learning within the given training time.

Our experiments on hyperparameter tuning showed that specific settings, such as a lower epsilon for exploration and a balanced reward structure, were essential for optimizing the agent's performance in flappy bird. These findings highlight that simpler tabular approaches, when feasible, can often be more practical than neural network-based methods in environments where the state space can be discretized effectively.

Overall, this project provided valuable insights into the strengths and limitations of Q-learning and DQN, underscoring that method selection should align with the environment's complexity, training resources, and desired outcomes.