Mastering Flappy Bird with Reinforcement Learning

Justin Li* zetianli@stanford.edu

Ziang Song* ziangs@stanford.edu

Essie Cao* essiecao@stanford.edu

Abstract

Reinforcement Learning has demonstrated remarkable success in game-playing. It even achieves superhuman performance in various environments. In this study, we applied Deep Q-Network(DQN) [1] to train an agent for an enhanced version of Flappy Bird. We included coin collection mechanism in order to make the decision-making process more challenging for the agent. We explored the impact of reward design, state representaion, and Best Experience Replay(BER) on training performance by comparing four model variants. Our findings suggest that minimizing unnecessary complexity in RL models can lead to faster convergence, improved stability, and higher scores. This study provides insights for optimizing deep RL models in constrained environments.

1 Introduction

Flappy Bird, a casual mobile game released in 2013, was created by Vietnamese game developer Dong Nguyen through his company, Gears. This side-scrolling game features a bird named Faby, which players navigate through gaps between green pipe columns without colliding. The score increases based on the number of pipes successfully passed. By the end of January 2014, it had become the most downloaded free game on the iOS App Store. However, on February 10, 2014, it was removed from both the App Store and Google Play due to concerns over its highly addictive nature and excessive usage. [2]

Figure 1: App icon

Reinforcement Learning (RL) provides an effective approach for training an AI agent to play Flappy Bird by learning optimal strategies through interaction with the game environment. The game can be modeled as a Markov Decision Process (MDP), where the agent observes the bird's position and velocity, selects actions (flap or do nothing), and receives rewards for successfully passing pipes.

Various RL methods have been applied to Flappy Bird, including Deep Q-Networks (DQN)[1] which estimate action values using a neural network, and Policy Gradient methods like Proximal Policy Optimization (PPO)[3], which directly optimize the decision-making policy. Evolutionary algorithms such as Neuroevolution (NEAT)[4] have also been used to evolve neural networks for better performance. Despite challenges like sparse rewards and exploration-exploitation trade-offs, RL enables agents to achieve superhuman performance in this seemingly simple yet complex game.

In this study, we developed a reinforcement learning (RL) agent using Deep Q-Networks (DQN) to play an enhanced version of Flappy Bird, incorporating a coin collection mechanism to introduce strategic decision-making. Through a series of experiments, we evaluated different reward structures and the impact of Best Experience Replay (BER) on training performance. Our results indicate that BER leads to more stable learning by refining policies over time, while

simpler reward designs and state representations improve learning efficiency and model performance. The findings suggest that minimizing unnecessary complexity in reward signals and state space can enhance generalization, leading to higher scores and better overall agent performance.

2 Related Work

Reinforcement learning (RL) has been widely applied in game-playing scenarios, demonstrating impressive capabilities in learning optimal strategies through trial and error.

Reinforcement Learning in Games AlphaGo Zero[5] was designed to train itself purely through self-play, without relying on any human-generated data. This autonomous learning capability allowed it to surpass all previous versions of AlphaGo and even the best human players in an unprecedentedly short period. Beyond board games, RL has also been applied to video games with high-dimensional visual inputs.DeepMind's pioneering work demonstrated that RL agents could learn to play Atari 2600 games[6] directly from pixel inputs, achieving superhuman performance in games like Breakout, Pong, and Enduro.Similarly, the ViZDoom platform[7] has enabled researchers to train RL agents in 3D first-person shooter environments, facilitating studies on navigation and combat strategies.

Q-learning and Deep Q-Networks (DQN) Q-learning[8], is a foundational reinforcement learning algorithm that enables agents to learn optimal action policies without requiring prior knowledge of the environment's dynamics. By iteratively updating a Q-table based on the Bellman equation, Q-learning efficiently learns optimal policies in discrete state-action spaces. However, traditional Q-learning struggles with large state spaces due to the necessity of storing and updating values for every possible state-action pair.

To overcome this limitation, deep reinforcement learning methods such as Deep Q-Networks (DQN) [1]were introduced by Mnih. DQN combines Q-learning with deep neural networks to approximate the Q-value function in high-dimensional environments. In their paper, DQN successfully achieved human-level performance on various Atari 2600 games by employing experience replay and target network stabilization. Given that Flappy Bird is an environment with high-dimensional state representations and continuous decision-making, we applied DQN in our Flappy Bird game as a promising approach for developing an agent that learns an optimal flight strategy.

Further enhancements, such as Rainbow DQN [9], combined various improvements including prioritized experience replay, dueling architectures, and distributional RL, leading to more robust performance in complex environments.

3 Approach

In this section, we introduce the Deep Q-Network (DQN) algorithm in Deep Reinforcement Learning and our key modifications to improve performance on the Flappy Bird environment.

3.1 Game Setting

- Frame rate of the game: $30\text{frames}/\text{second}$
- Bird's horizontal speed: $5px/\text{frame}$
- Time between each state: $\Delta t = t_{n+1} - t_n = \frac{1}{30}$ seconds
- Constant changing vertical velocity when choose no tap: $v_{t+1} - v_t = -1$
- Gap between each pair of upper & lower pipes: $120px$

3.2 Integration of Coins into the Flappy Bird Model

We added a new feature to the game: coins. This makes the game more interesting and complex. Collecting each coin gives a 5 bonus score, and the coins are generated randomly.

Risk-Reward Tradeoff Without coins, the goal is maximizing survival, so the best action is always deterministic. As shown in the Figure 2, when there are coins at random location, the agent faces a tradeoff between avoiding obstacles when maximizing survival and trying to collect coins for extra rewards but increasing the probability of crashing. In this case, a deterministic policy may not always be optimal:(1)If a coin is in a risky position, it might sometimes be worth going for it and sometimes not, depending on the current game state and the probability of survival. (2)The agent might want to randomly attempt risky moves with some probability, making the policy stochastic.

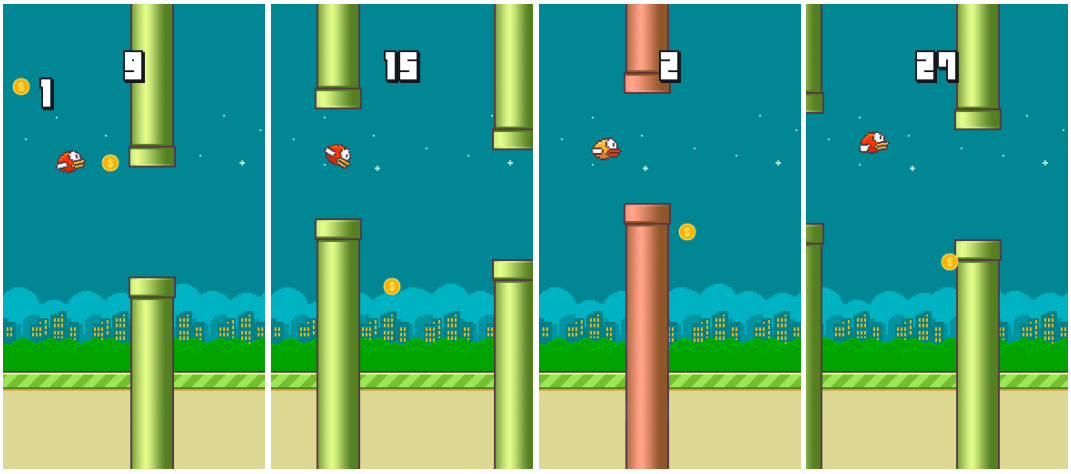

Figure 2: Coin Location Examples (1. Low Risk, 2. Low to Medium Risk, 3. High Risk, 4. Crashing)

3.3 MDP Modeling

3.3.1 State Space

$$S = \{(h, p^x, p^y, c^x, c^y, v^y, score) \mid (h, p^x, p^y, c^x, c^y, v^y, score) \in \mathbb{R}^7\} \cup \{\text{(score, dead)}\}$$

where

- h: height of the bird to the ground
- p^x: horizontal distance to the gap of midpoint for next pipe
- c^x: horizontal distance to the next coin
- p^y: vertical displacement to the gap of midpoint for next pipe (upward +)
- c^y: vertical displacement to next coin (upward +)
- v^y: vertical velocity (upward +) of the bird
- $score$: current score of the game, can consider as the tuple (pp, cc) to record the total scores where pp is the number of pipes passed and cc is number of coins collected.
- $dead$: indicates terminal state, the bird hitted the pipe or touched the ground

3.3.2 Transitions

Transitions occur from the state in the first frame to the state in the second frame, with transition probabilities dependent on game settings (such as the distribution for generating coins and pipes).

3.3.3 Action Space

$$A = \{0, 1\}$$

where flap is labeled by 1 and no flap is labeled by 0.

3.3.4 Rewards

Here we provide a simplest example. For non-terminate state s_t:

$$R(s_t, a_t, s_{t+1}) = \begin{cases} 5 & \text{if score}_{t+1}.cc - \text{score}_t.cc = 1 \text{ (collected a coin)} \\ 1 & \text{if score}_{t+1}.pp - \text{score}_t.pp = 1 \text{ (passed a pipe)} \\ -10 & \text{if } s_{t+1} = dead \end{cases}$$

Note: The reward design could be more complicated, will show it in the later part.

3.3.5 Goal

The goal is to maximize the score over time steps t from 1 to 900 and 1800 (30 frames per second for 30 and 60 seconds), and also make sure the bird survive as long as possible when there is time constraints.

3.4 Online-DQN

Deep Q-Network (DQN) is a reinforcement learning algorithm that combines Q-learning with deep neural networks to learn optimal action-value functions. The algorithm maintains two neural networks - a policy network that is actively trained, and a target network that is periodically updated to provide stable training targets.

We choose the Online-DQN method for three main reasons: (1)The state space in our problem is very large, making it more efficient to use neural networks (NN) to approximate Q-values rather than relying on tabular Q-learning. (2) Our action space is quite simple, consisting only of "flap" or "no flap". Therefore, there is no need to use more complex algorithms like Proximal Policy Optimization Algorithms for such a small action space.(3) Easy to interact with game during training, learn along the way instead using human play data, which can save a significant amount of time to get the data.

Algorithm 1 Deep Q-Network (DQN) Algorithm (Online)

Initialize replay buffer $\mathcal{D}$
Initialize policy network $Q(s, a; \theta)$ with random weights θ
for each episode **do**
    Initialize state s_1
    for each step in episode **do**
        With probability ϵ select a random action a_t
        Otherwise select $a_t = \arg\max_a Q(s_t, a; \theta)$
        Execute action a_t and observe reward r_t and next state s_{t+1}
        Store transition (s_t, a_t, r_t, s_{t+1}) in $\mathcal{D}$
    end for
    Sample random mini-batch of transitions (s_j, a_j, r_j, s_{j+1}) from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} Q(s_{j+1}, a'; \theta) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(s_j, a_j; \theta))^2$ with respect to θ
end for

The policy network $Q(s, a; \theta)$ takes a state s as input and outputs Q-values for each possible action a. During training, experiences (s, a, r, s') are collected and stored in a replay buffer. Mini-batches are sampled from this buffer to update the current network parameters θ using the following loss:

$$L(\hat{\theta}) = \mathbb{E}_{(s,a,r,s')}[(r + \gamma \max_{a'} Q(s', a'; \theta) - Q(s, a; \hat{\theta}))^2]$$

where $\hat{\theta}$ are the parameters of the target network. The target network parameters are periodically copied from the policy network to maintain stability.

3.5 Best Experience Replay (BER)

A key modification we make to the standard DQN algorithm is the introduction of a "best experience" replay buffer. Rather than only storing recent experiences, we maintain a separate buffer that specifically saves experiences from episodes that achieve high scores. When sampling batches for training, we combine experiences from both the standard and best experience buffers.

This approach helps preserve and learn from successful trajectories, allowing the agent to more effectively learn behaviors that lead to high rewards. The best experience buffer is updated whenever an episode exceeds the current best score, replacing older experiences with the new successful trajectory.

3.6 Reward Design

We found that the results are highly dependent on the reward design. In addition to giving rewards for passing a pipe or collecting coins, we also considered the following modifications to improve the policy:

Survival reward Every frame the bird survives, it receives a small positive reward.

Distance-based reward Each time the bird dies, in addition to a large negative reward, it also receives a positive reward based on how close it was to passing the pipe.

Flap penalty Each time the bird flaps, it receives a small negative reward. This discourages the bird from flapping too frequently, as a human player cannot flap 10 times per second.

4 Experiments

In this section, we provide a comprehensive overview of our experimental setup and methodology.

4.1 Game Modifications and RL Agent Implementation

We utilized the FlapPyBird game [10], implemented using the Pygame library[11]. The following modifications were made to gather the necessary data for training our DQN model.

Coins To enhance the complexity and engagement of the game, we introduced a coin collection feature. Each coin collected awards the player an additional 5 points, with no penalty for missing coins. Coins are generated every 3 seconds (equivalent to 90 frames) with a probability of 0.5. Their positions are uniformly distributed within a rectangular area between two pipes. This addition introduces strategic decision-making, as players must weigh the risk of collecting high-reward coins against maintaining safety.

RL Agent We developed a reinforcement learning agent, referred to as 'DQNagent', to autonomously play the game. This agent interacts with the game using Pygame and is capable of: (1) Automatically playing the game using a policy derived from a neural network-approximated Q-function. (2) Recording gameplay history in a replay buffer, essential for the DQN algorithm.(3) Utilizing the stored experiences in the buffer to refine its policy through Q-learning every 10 episodes.

4.2 Evaluation Metrics

We employ specific metrics to assess the performance of our reinforcement learning model in playing the Flappy Bird game. These metrics provide insights into the learning progress, stability, and efficiency of the agent.

- **Episode Score**: The episode score represents the total reward accumulated by the agent within a single episode.
- **Rolling Mean Score**: The rolling mean score is computed by averaging scores over a fixed number of recent episodes. This metric provides a clearer indication of long-term learning trends.
- **Max Score So Far**: The max score so far tracks the highest episode score achieved at any point during training.
- **Coins Collected**: The number of coins collected per episode evaluates the agent's ability to balance risk-taking with score maximization.

To gain deeper insights into the strengths and limitations of our model, we further analyze its performance through additional comparisons. We compare the model's learning progress with and without the best experience replay mechanism to evaluate its impact on training stability and overall performance. Additionally, we investigate how different reward structures affect the agent's learning behavior.

4.3 Experimental Details

Our experiment utilizes a neural network architecture optimized for the Flappy Bird environment. We employ the AdamW optimization algorithm [12] with a learning rate of 0.001 to minimize the mean squared error (MSE) loss during training updates. The key components of our experimental setup are:

Network Architecture The neural network architecture used in this reinforcement learning model consists of an input layer with six neurons, each corresponding to a specific dimension in the state space. The network includes a single hidden layer with 128 neurons activated using the ReLU function, allowing the model to learn non-linear representations effectively. The output layer consists of two neurons, representing the Q-values for the two available actions: "flap" and "no flap."

Experience Replay To enhance learning efficiency, an experience replay mechanism is implemented. The replay buffer has a capacity of 10,000 frames, ensuring stable learning by preventing the network from overfitting to recent experiences. Additionally, a separate "best experience" buffer is maintained to store highest-scoring trajectories, allowing the agent to learn from successful gameplay. Mini-batches of size 256 are sampled from these buffers to perform gradient updates, ensuring a more effective training process.

Training Parameters The model follows specific training parameters to optimize learning. A discount factor (γ) of 0.99 is used to weigh future rewards, encouraging long-term decision-making. To balance computational efficiency, training is performed every 10 episodes, reducing the computational overhead.

Exploration Strategy For exploration, an ϵ-greedy strategy is employed. Initially, the value of ϵ is set to 0.1, allowing the agent to explore the environment sufficiently. An exponential decay rate of 0.995 is applied per episode to gradually shift the model from exploration to exploitation. Eventually, ϵ is reduced to a minimum value of 0.0, ensuring that the agent relies entirely on its learned policy. To further enhance exploration during training, noise is added to action selection, encouraging diverse decision-making and preventing convergence to suboptimal policies.

4.4 Training on Different Model Variants

We designed four model variants to systematically study the impact of reward design, state representation, and experience replay on agent performance. DQNBasic serves as the baseline, using a simple reward structure focused solely on survival. DQN extends this by incorporating additional distance-based rewards, aiming to guide the agent toward coins. DQNBest_tj builds on DQN by integrating Best Experience Replay (BER), which prioritizes impactful learning experiences to improve stability and exploration. Finally, DQNSimple combines BER with a simplified reward and reduced state space, allowing us to assess how minimal yet focused design can lead to more efficient and effective learning.

4.5 Training Results

From the above setting, we run train the **Online-DQN** for 1000 episodes. The followings are the training results based on the different reward design and modelling.

4.5.1 Performance of DQNBasic

Here we applied the most basic reward design as the reward function in **3.3.4**, denote the model under this setting as `DQNBasic`

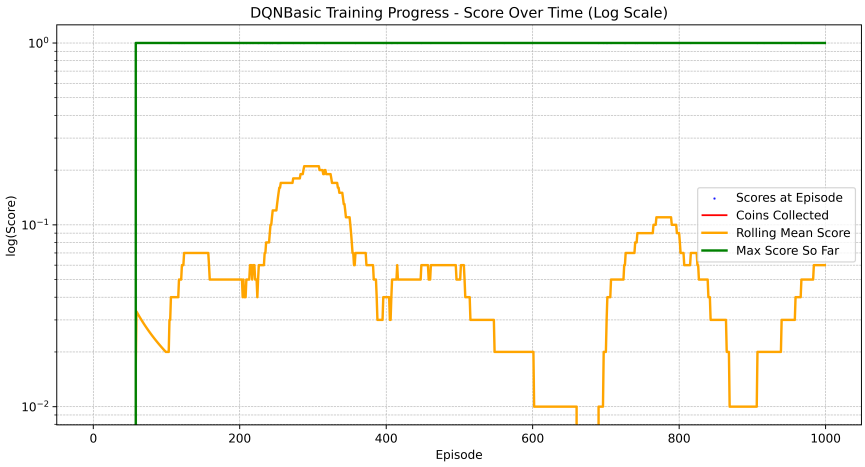

Figure 3: Training Result for `DQNBasic`

This plot shows the training progress of the DQNBasic agent, revealing its struggles due to the simplistic reward structure:

- **Limited Progress:** The agent never collects coins and rarely passes pipes, as indicated by the consistently low rolling mean score.
- **Max Score Plateau:** The highest score remains very low (≤ 10), showing that the agent fails to achieve meaningful progress.
- **Sparse High Scores:** The individual scores per episode show little variance, suggesting the agent doesn't explore effective strategies well.

The basic reward setting does not provide enough incentive for the agent to improve significantly, leading to poor learning and an inability to navigate past pipes effectively.

4.5.2 Performance of DQN

Based on the reward in `DQNBasic`, we applied all the additional reward designs described in **3.6**, denote the model under this setting as `DQN`. The details of additional reward design will be the following, for non-terminal s_{t+1}:

$$R(s_t, a_t, s_{t+1}) = \underbrace{0.03}_{\text{survival reward}} - \underbrace{0.3 \cdot a_t}_{\text{flap penalty}} + \underbrace{0.002 \cdot \mathbb{I}(|p^y_{t+1}| \leq 150) + 0.001 \cdot \mathbb{I}(|c^y_{t+1}| \leq 60) + 0.001 \cdot \mathbb{I}(c^x_{t+1} \leq 60)}_{\text{distance based reward}}$$

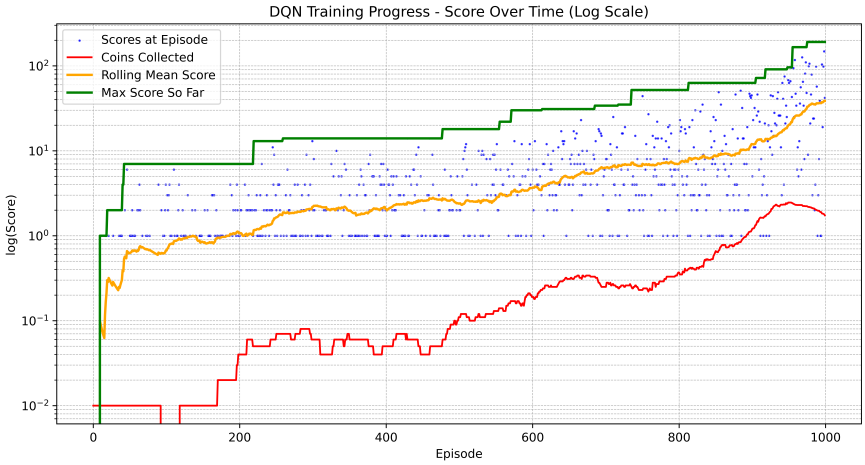

Figure 4: Training Result for `DQN`

This plot represents the training progress of the DQN agent, showing substantial improvements over time after adding additional rewards:

- **Consistent Learning Progress:** The rolling mean score increases steadily, demonstrating that the agent is effectively learning to navigate the environment.
- **Higher Max Score:** The agent progressively achieves new high scores, which is approximately 400, suggesting successful policy improvements.
- **Coins Collected:** The agent is progressively collecting more coins, signifying that it is optimizing not just for survival but also for rewards.
- **Densely Packed Blue Dots at Higher Scores** More episodes are reaching higher scores, showing improved consistency in agent performance.

This DQN implementation is significantly stronger than the basic versions, showing steady improvements in score, stability, and coin collection.

4.5.3 Performance of DQNBest_tj

On the basis of the above `DQN` model, we added **Best Experience Replay**, denote the model under this setting as `DQNBest_tj`.

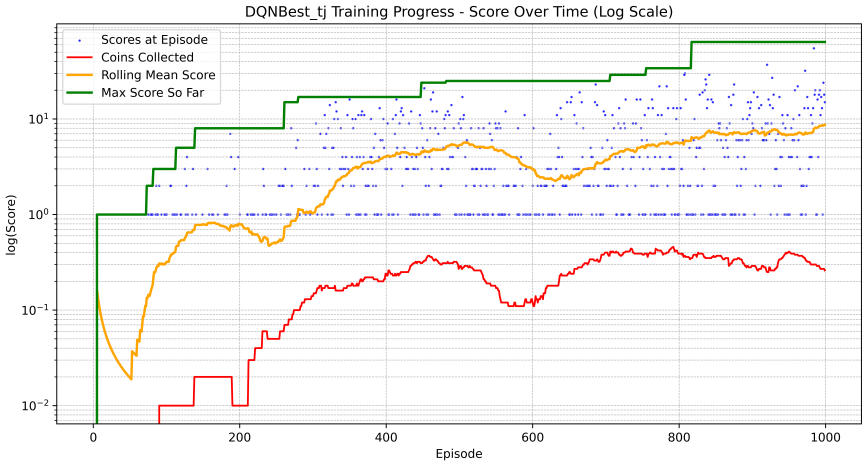

Figure 5: Training Result for `DQNBest_tj`

This `DQNBest_tj` training plot also shows notable improvements:

- **High Max Score** The maximum score continues to increase, indicating that the agent is consistently breaking past previous performance ceilings.
- **Consistent High Scores:** Unlike previous runs where scores were more spread out, here the agent consistently achieves higher scores, reflecting improved stability.

The addition of best history replay has clearly stabilized learning, improved score consistency, and helped sustain performance gains. Future improvements could focus on fine-tuning reward structures or exploration strategies to push the agent further.

4.5.4 Performance of DQNSimple

On the basis of the above `DQNBasic` model, we added **Best Experience Replay**, and also modified the state space and additional reward. Denote the model under this setting as `DQNSimple`, and the details of state space and additional reward will be the following

$$S = \{(p^x, p^y, v^y, score) \mid (p^x, p^y, v^y, score) \in \mathbb{R}^4\} \cup \{\text{(score, dead)}\}$$

and

$$R(s_t, a_t, s_{t+1}) = \underbrace{0.03}_{\text{survival reward}} \underbrace{- 0.3 \cdot a_t}_{\text{flap penalty}}$$

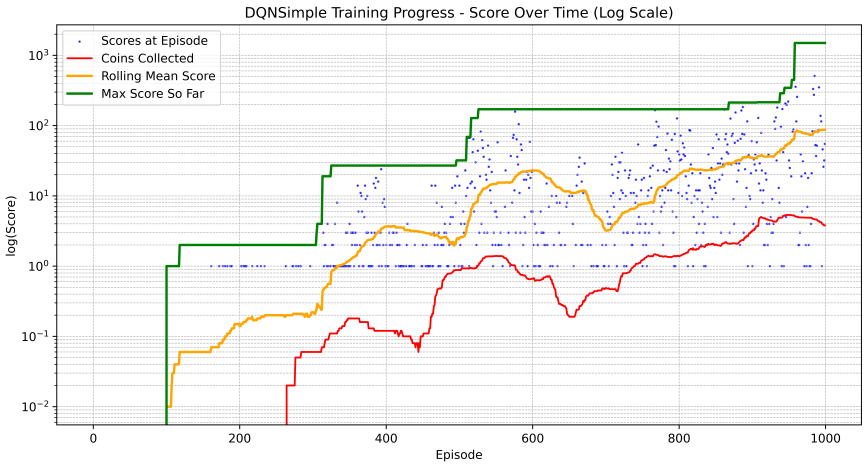

Figure 6: Training Result for `DQNSimple`

This DQNSimple training plot exhibits solid learning progress with notable improvements in performance:

- **Strong Performance Growth:** The rolling mean score shows a steady increase, indicating effective learning. The agent progressively adapts to the environment and achieves higher scores over time.
- **Higher Max Score:** The maximum score reaches to around 1500, suggesting that the agent is occasionally making breakthroughs in performance.
- **Improved Coin Collection:** Compared to previous models, the agent learns to collect more coins and maintains a strong upward trend. This suggests better reward optimization.
- **High Variability in Scores:** The increasing spread of blue dots at higher score levels reflects greater variability in episode performance, meaning the agent is capable of achieving strong results but still exhibits some inconsistency.

This DQNSimple model shows strong learning dynamics, achieving higher scores, better coin collection, and steady performance growth. However, some instability and plateauing suggest room for fine-tuning—potentially in exploration strategy or reward adjustments—to ensure continued performance improvements.

4.6 Training Result Summary

4.6.1 Impact of Best Experience Replay on Model Training

We compare the performance of DQNBasic and DQNBest_tj, which share the same reward design, with DQNBest_tj incorporating Best Experience Replay. Our key observations are:

Exploration & Periodic Improvements The `BER` can periodically tests new actions and strategies, which may initially decrease performance before finding a better policy. This creates a wave-like pattern, where improvement happens after a temporary dip in performance. This is likely due to continued policy refinement and controlled exploration.

More stable learning The rolling mean score in `DQNBest_tj` is smoother and has less variance than `DQN`, meaning learning is more stable when using BER, which indicating a more controlled and structured learning process.

4.6.2 Impact of Reward Design on Model Performance

We analyzed how different reward designs influenced performance particularl comparing DQNBest_tj and DQNSimple, which differ only in reward structure. Our key observations are:

The simpler reward design the better? The training result indicates the **reward simplicity drives higher scores**. The key reason `DQNSimple` achieves higher scores is likely due to its simplified reward structure—it only rewards survival and does not include an additional reward for "getting closer to a coin." In this game, surviving longer naturally leads to higher rewards, making intermediate incentives unnecessary. In contrast, `DQN` may encourage the agent to take suboptimal paths by rewarding proximity to coins, which could cause risky movements or distractions that reduce overall survival time. By removing unnecessary intermediate rewards, `DQNSimple` ensures that the agent focuses entirely on staying alive, which ultimately translates into better long-term performance and higher scores. This reinforces the idea that simpler, direct reward signals often lead to more effective learning and superior results in reinforcement learning.

The less state the better? Another key factor contributing to `DQNSimple`'s superior performance is its simpler state space—it uses only 4 state variables, whereas `DQN` relies on 7. A smaller state space makes learning more efficient because the agent has fewer dimensions to process and optimize, reducing the complexity of the Q-function approximation. `DQN`'s larger state space may introduce unnecessary noise or redundant information, making it harder for the model to generalize and converge to an optimal policy. Since survival is the most important factor in this game, `DQNSimple`'s streamlined state representation ensures that the agent focuses only on the most critical variables, leading to faster learning, better generalization, and ultimately higher scores. This further supports the idea that simpler models, when designed with the right core elements, can outperform more complex ones.

5 Analysis

5.1 Testing In Game Environment

From the training result, the `DQNSimple` seems have dominance performance among other due to its simple reward design and state space. However, `DQNSimple`'s higher scores might just caused by its targeted long survival time in

the additional reward design, because the long survival time usually result the higher scores. In this case, we will do analysis on the performance of `DQN`, `DQNBest_tj` and `DQNSimple` on a small fixed period of time.

We use the `DQN`, `DQNBest_tj` and `DQNSimple` models to **play** the game for for 300 episodes and terminate when reach to 900 and 1800 frames (30 seconds and 60 seconds). The followings are the game results

5.1.1 Game Results

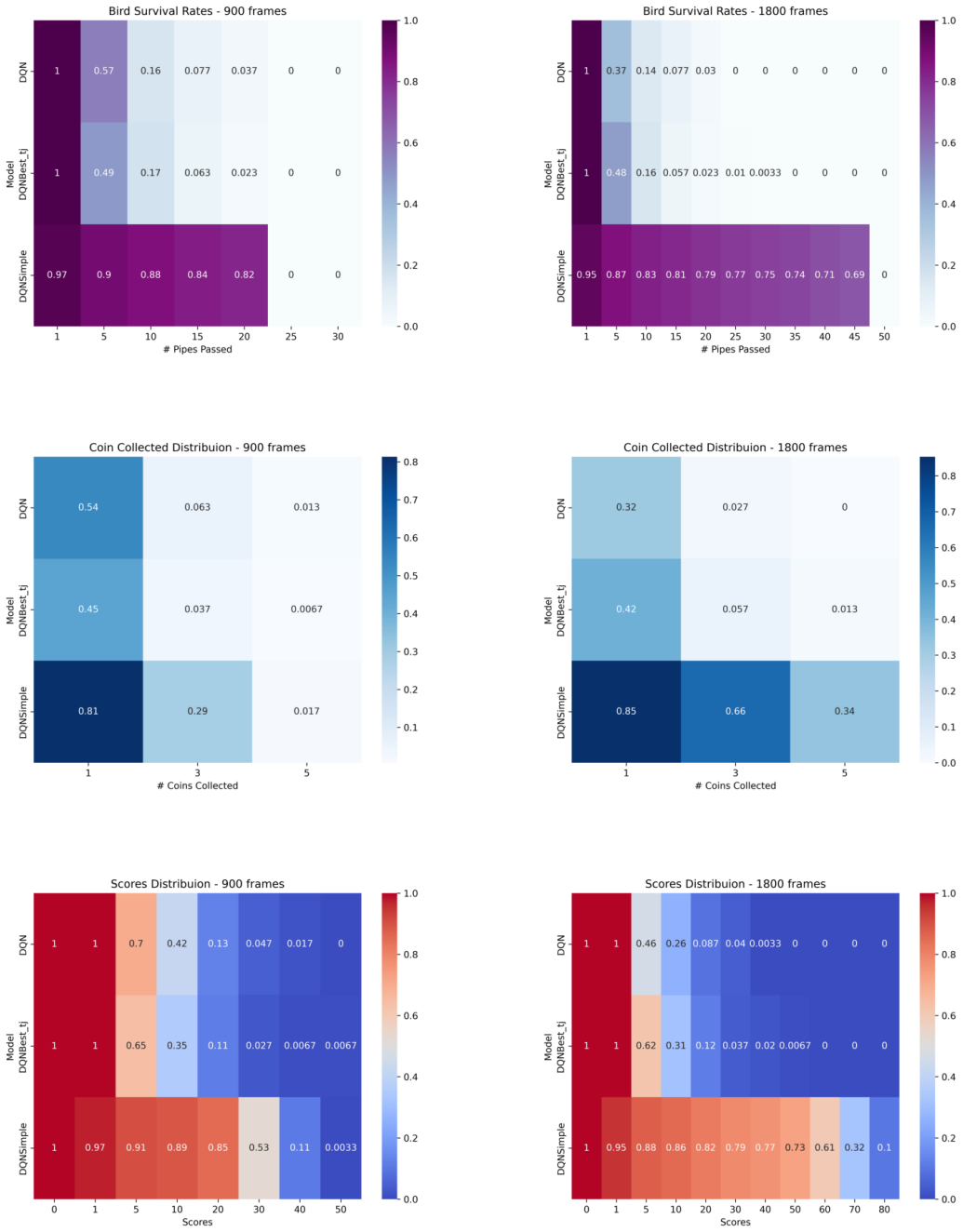

Figure 7: Game Result Survival Rates, Coins Collected, Scores

`DQNSimple` consistently outperforms `DQN` and `DQNBest_tj` in terms of survival rate, score distribution, and coin collection across both 900 and 1800 frames. Its higher survival rates indicate that it prioritizes longevity, which naturally

leads to higher scores and better performance in coin collection. In contrast, DQN and DQNBest_tj exhibit significantly lower survival rates, limiting their ability to accumulate high scores or collect rewards efficiently. The results suggest that DQNSimple's simpler reward design and state space contribute to its dominance by encouraging strategies that maximize survival and exploration.

However, it remains unclear whether its superior performance is due to an optimal decision-making policy or merely an alignment with the reward structure that favors extended survival. Further investigation is required to determine whether DQNSimple generalizes well beyond this specific setup or if its advantage is primarily a result of reward shaping.

5.2 Differences Between Agent and Human Players' Behavior

The behavior of DQN, DQNBest_tj and DQNSimple differs from human players primarily in survival strategies and risk management. The agents prioritize survival and reward maximization based on the reinforcement learning framework, whereas human players might adopt more adaptive strategies influenced by experience, intuition, and reaction speed.

Exploration & Exploitation Agents follow learned policies, which can sometimes be overly optimized for the training environment, whereas humans might explore new strategies dynamically.

Mistake Recovery Humans can anticipate failures and adjust immediately, whereas agents follow predefined learned behaviors, making them more vulnerable to unexpected scenarios.

5.3 Convergence

Convergence analysis assesses whether the learning process stabilizes over time and if the models reach an optimal policy.

Performance Stability DQNSimple appears to have converged towards a stable strategy that emphasizes survival and reward collection. The DQN and DQNBest_tj show fluctuations during game playing and training, indicating that they might not have fully converged or have learned suboptimal policies.

Convergence Speed DQNSimple's simpler reward structure may have led to faster convergence since it prioritizes long-term survival without overcomplicated the decision-making. DQN and DQNBest_tj, having more complex architectures and possibly more intricate reward functions, might require more training iterations to fully stabilize.

Potential Overfitting If DQNSimple is too optimized for survival in a specific setting, it may not generalize well to new or slightly modified environments (e.g. more complicated FlappyBird game), raising questions about the robustness of its learned policy. Additional training data or different exploration strategies might be needed to ensure generalization beyond the current test conditions.

6 Conclusion and Future Work

Our results demonstrates that **DQN** is capable of learning effective policies in environments with large state spaces, but in some cases, simpler DQN variants, like DQNSimple, can outperform more complex models by prioritizing survival and reward accumulation. The results indicate that a well-designed reward structure and state representation can lead to faster convergence and more stable policies, as seen in DQNSimple, which achieved consistent and high performance more quickly than the other models. This suggests that in certain scenarios, a simpler model with a well-aligned reward design can be more efficient and robust than a complex one. Furthermore, this analysis highlights the effectiveness of reinforcement learning (RL) in gaming applications, as the trained agents successfully learned strategies to maximize survival and score. RL-based agents can exceed human performance in structured environments, providing valuable insights into AI-driven decision-making and autonomous learning.

While the current models demonstrate noteable performance, further research is needed to improve generalization and adaptability. One important direction is exploring how well the learned policies transfer to new game variations, different reward structures, or unseen environments. Moreover, a more in-depth comparison between agent behavior and human decision-making may provide insights into strategic differences, particularly in reaction patterns and risk assessment.

Another key area of improvement lies in efficiency and convergence analysis. The observation that simpler models like DQNSimple converge faster and more stably suggests that a well-designed reward structure can play a crucial role in

effective learning. Future work could explore how to leverage this insight to improve training efficiency, performance or model robustness in complex environments by incorporating techniques like transfer learning, curriculum learning[13], or alternative architectures such as PPO[3] or DoubleDQN

Team contributions (Required for multi-person team)

Justin Li Worked on the introduction, **3.1**−**3.3.5**, **4.4**−**6** and the corresponding code implementation.

Ziang Song Coding everything except evaluation. Work includes modify the initial game to enable coin generation and RL agent to interact with the game and DQN. Wrote Section 4.3-4.5 and Section 5.1-5.3 in the report.

Essie Cao Worked on Abstract, modified 1, worked on section 2, 4.2, 4.4.

References

[1] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[2] Wikipedia contributors. Flappy Bird, 2025. Retrieved March 17, 2025.

[3] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[4] Kenneth O. Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10(2):99–127, 2002.

[5] David Silver, Julian Schrittwieser, Karen Simonyan, et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354–359, 2017.

[6] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, et al. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[7] Michał Kempka, Marek Wydmuch, Grzegorz Runc, et al. Vizdoom: A doom-based ai research platform for visual reinforcement learning. In *IEEE Conference on Computational Intelligence and Games (CIG)*, pages 1–8, 2016.

[8] C. J. C. H. Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8:279–292, 1992.

[9] Matthias Hessel, Joseph Modayil, Hado van Hasselt, et al. Rainbow: Combining improvements in deep reinforcement learning. In *AAAI Conference on Artificial Intelligence*, 2018.

[10] Sourabh Verma. Flappybird: A flappy bird clone using python-pygame, 2025. GitHub repository.

[11] Pygame Community. Pygame: A python library for game development, 2025. Version 2.6.0.

[12] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017.

[13] Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. *Proceedings of the 26th International Conference on Machine Learning (ICML)*, pages 41–48, 2009.

A Appendix

Project Github Link: https://github.com/sza919/Flappy_bird_RL_coins