**Github link:** [**https://github.com/khadijayadi/flappy-superhero-rl**](https://github.com/khadijayadi/flappy-superhero-rl)

**Game\_sreenrecord:**

[**https://drive.google.com/file/d/1j5S8gz\_D2ItfOuUHo4ZcQJMqoI10e1QS/view?usp=sharing**](https://drive.google.com/file/d/1j5S8gz_D2ItfOuUHo4ZcQJMqoI10e1QS/view?usp=sharing)

A close up of a logo

Description automatically generated

**Assessment Submission Form**

|  |  |
| --- | --- |
| **Student Number**  (If this is group work, please include the student numbers of all group participants) | GH1025231 |
| **Assessment Title** | **Reinforcement Learning for Flappy Superhero Game** |
| **Module Code** | B143 |
| **Module Title** | AI Studio |
| **Module Tutor** | Amirhossein Jamalian |
| **Date Submitted** | 01.07.2025 |

|  |
| --- |
| **Declaration of Authorship**  I declare that all material in this assessment is my own work except where there is clear  acknowledgement and appropriate reference to the work of others.  I fully understand that the unacknowledged inclusion of another person’s writings or ideas or works in  this work may be considered plagiarism and that, should a formal investigation process confirms the  allegation, I would be subject to the penalties associated with plagiarism, asper GISMA Business  School, University of Applied Sciences’ regulations for academic misconduct.  Signed …… … … …Khadija ayadi … … … … … … … … … … … … . Date 01.07.2025 |

**Reinforcement Learning for Flappy Superhero Game**

1. **Introduction:**

In machine learning, reinforcement learning involves agents learning how to make choices by interacting with their environments and working to gradually maximize some scalar reward signal. Unlike traditional supervised learning methods, RL agents learn not from labelled data, but from their own actions by figuring out the consequences of these actions, adjusting their strategies accordingly. This is suited to dynamic environments like games where choices must be made sequentially and rewards are often delayed.

This project is about developing a customised version of the classic Flappy Bird game and modifying it for reinforcement learning. The bird in this Flappy bird game is replaced by a superhero character who must take the character through obstacles (buildings) while avoiding collisions. Through a sufficiently regulated but hard environment, the RL agent tries to learn a policy through trial and error.  
  
Our reinforcement learning agent was trained using the Q-learning algorithm, which is a basic and widely used RL algorithm. Q-learning is a model-free method, which implies that our agent will learn to value state-action pairs after many episodes while gradually improving its performance in the environment. This is achieved without a model describing its dynamics.  
  
One of the main attractions of taking on this project was the simple game mechanics of Flappy Bird, but the complexities it represents in the learning process. Tuning rewards , state representation, and learning parameters , to allow the agent converge to a good policy , is a bit of a challenging act. Through this project, we focused on tabular Q-learning deliberately, avoiding the use of deep neural networks and ultimately demonstrating that even simple algorithms can learn good behaviours with a well designed environment.  
  
This report provides a complete analysis of the pipeline of all the work we've done to develop an RL agent, including designing the game and environment, training/evaluation, and eventually playing and visualizing the learned behaviour of the agent. We would like to reflect on our iterative debugging and improvements in policy shaping and training that resulted in a competent and effective Flappy Superhero agent.

1. **Game design:**

Flappy Superhero is built as a basic 2D side-scrolling game inspired by the foundational ideas of Flappy Bird. The player controls the superhero as it avoids colliding with pipes and, ideally, stays alive for as long as possible. The game was built entirely in Pygame, and has a structure similar to the OpenAI Gym style with functions such as reset(), step(action), and render() to manage the agent.

A screenshot of a computer

Description automatically generatedA screenshot of a video game

Description automatically generated  
**Screenshot of the game window**

**Key Features:**

- Custom Pygame Environment: A game universe built entirely from scratch in Python with Pygame.  
  
- Superhero Character: The bird has been changed to a graphic of a superhero taken from assets.  
  
- Passing Pipes: The agent must learn to flap at the correct time in order to navigate gaps between the vertical pipes.  
  
- Scoring system: Everytime the agen passes through a pipe he gets a +1 in the score that is - being displayed on screen.  
  
- Reward logic: The agent receives positive rewards for staying alive and passing pipes.   
 The agent receives negative rewards for crashing.   
  
- Collecting The Coin: when The agent reaches 10 points, a coin appears between the pipes. When the agent collects the coin, the game ends in a win.  
  
- Game end message: When the agent collects the coin a "Congratulations" message appears in the game window and the episode ends.

1. **Environment Design**:

The environment is created in which a superhero agent needs to fly between the columns of pipes that are aligned vertically. The environment behaves like OpenAI Gym environments as we discussed earlier, because we too have built in the same methods: reset(), step(), render(), and get\_state(), which represent the agent interaction loop. The advantage of setting it up this way is that it will work with standard reinforcement learning methods like just using tabular Q-learning. In this section, we will be reviewing the flappy\_superhero\_env.py file.

On construction, the environment initializes the game window, as shown in the **figure 1**, the graphic assets (background, superhero, and coin), and declares all constant properties such as the screen width, screen height, gravity, flap power, and column properties. When render\_enabled=True, it will enable graphical rendering to give the agent visual feedback.

A screen shot of a computer program

Description automatically generated

Figure 1 : Environment Initialization & Assets Setup

The game state, which includes coin status, score, pipe location, vertical position, and bird's velocity is reset by the reset() method. Restarting a new episode is done in this way as well.

Physics and the logic of the game will be taken care of using the function step(action). Velocity in the vertical axis will be updated, pipes will be moved to the left, the location of the bird will be updated, and benefited from ability to detect collisions based on agent state.

There is 2 actions either flap (action = 1) or do nothing (action = 0) . The reward system , see the **figure 2** below , works as the following : When an agent passes a pipe, it receives a +100 reward , if the agent survives a number of steps, the agent receives a +1 survival bonus. There is a proximity bonus of up to +5 based on how close he is to the center of the pipe gap; this bonus rewards precision. In contrast, a collision with a pipe or flying off the top or bottom of the screen incurs a -1000 penalty which ends the episode. when he sucess to collect the coin the agent receives a +1000 reward .

A screenshot of a computer program

Description automatically generated

Figure 2 : Game Logic and Rewards allocation

Simulating the environment dynamics, check\_collision() determines if the bird has collided with a pipe or is off-screen. The get\_state() method returns a discretized representation of the game state which includes the horizontal distance to the next pipe gap, vertical distance to the next pipe gap, and vertical velocity for the current frame. This is a compact tuple based representation of the agent’s state, which is helpful for tabular Q-learning.

Finally, the render() function draws the background, pipes, superhero, and current score and handles victory conditions, which involves collecting the coin and displaying the win message, which means that the agent has successfully mastered the task .

1. **Q-learning Implementation :**

The reinforcement learning agent is trained using the Q-learning algorithm written in q\_learning\_agent.py file . This method made the agent learn an optimal policy to play the game by updating a Q-table that maps state-action pairs to expected future reward. In this implementation, the agent interacts with FlappySuperheroEnv, getting observations from discrete state space, which consists of the horizontal distance to the pipe, the vertical distance to the gaps in the pipe, and the agent's vertical velocity. The agent gets to execute either flap, or do nothing based on the stored values of the Q-table .

**Training hyperparameters:**

**Episodes:** 50,000 game episodes used for training

**Gamma (γ) :** 0.99 , this is the discount factor that balances immediate and overall reward

**Alpha (α) :** a dynamic learning rate with a minimum learning rate of 0.01

**Epsilon (ε) :** exponentially decayed from 1.0 to a minimum of 0.05

🡪 We decided to use a high epsilon value to grant the agent more freedom for exploration in the early stages of learning, and ultimately better exploitation later on.

Every episode's step contributed to updating the Q-table using the Bellman equation. The Q-table was implemented as a Python dictionary where its keys tuples represent states and the values of these keys are NumPy arrays, which represent the action values. The Q-table is saved to a file q\_table.pkl using the pickle module. This allows for checkpointing and reloading Q-table for the next step of training and we will also be using it for evaluation. Intermediate checkpoints were saved every 1,000 episodes and the final Q-table was saved when the training was complete.

**How we got to the final optimal hyperparameters ?**

To evaluate the learning behavior of the agent, we ran two principal trials varying the epsilon decay method. On the first trial, we applied an approach with linear epsilon decay, with an initial epsilon = 1.0, multiplied by epsilon\_decay = 0.998 for each episode. With only 5000 episodes, the agent's exploration occurred too fast ,as we can see in **figure 3** , the epsilon falls below the minimum threshold of 0.05 before the end of the first 600 episodes, resulting in unstable learning and inferior policies, even if some rewards seemed slightly positive.

A screen shot of a number

Description automatically generated

Figure 3: Failed training printed outcomes

In the second trial, we changed the method to use an exponential decay formula, with EPSILON\_START = 1.0, EPSILON\_MIN = 0.05, and a slower DECAY\_RATE = 0.00005 for 50,000 episodes, as noted in **figure 4,** allowing the agent to explore freely before returning to achieve more stable and exploitative behavior. This was an important change to the training method that allowed me to produce long-run learning results and stable reward convergence behavior. The difference between the two runs is clearly seen in the screenshots of the printed terminal outputs of each run.

A screenshot of a computer

Description automatically generated

Figure 4 Final training printed outcomes

1. **Evaluation & Experimentation :**

After developing the game environment and the Q-learning logic I was then able to complete some training experiments, during which I recorded what the agent was doing when it was trained , through play\_with\_agent.py, allowing me to assess its performance and adjust the agent accordingly.

Early on, I was approaching Q-learning with shorter training runs and a few different hyperparameters as explained in the Q-learning implementation part on top, which produced some earlier training plots that are shown in the **figure 5** below .

A graph of different types of graphs

Description automatically generated with medium confidence

Figure 5 : First Training metrics

Those plots showed that although the agent occasionally reached positive rewards, the agent’s learning towards value function was not particularly stable, and in turn not consistent, with rewards significantly floating and not converging to an average, as well as with scores per episode that were also floating and remaining low.

These incremental improvements ultimately led me to optimize the Q-learning settings, including increasing the number of episodes to 50k, changing the epsilon decay to decrease gradually to stabilize the exploration versus exploitation more clearly over time, and revisiting the reward function within the environment. The plot in the file training\_curves\_log\_scaled.png, shown in **figure 6** , demonstrates the output of the improved configurations.

A graph and a chart

Description automatically generated with medium confidence

Figure 6 : Final Training metrics

In this plot I used a log scale on the y-axis to incorporate large swings of the reward signal over thousands of episodes whilst plotting the trends of the agent's learning over time, and the moving average line shows, without question, that the agent did discover a more stable and higher performing policy to some degree.  
  
Overall, these plots allow you to see some of the steps of progress the agent made throughout the training process, and it is clear that progressive experimentation was important in this reinforcement learning case. The plots capture checkpoints during the process and reveal that careful setting of the epsilon decay, reward shaping and training time significantly impacted the improved final plots, .  
  
In summary, the agent learned how to successfully operate in the Flappy Superhero environment with increased learning through an endless series of episodes of experimentation, largely adjusting the reward functions, hyperparameter tuning and volume of interaction with the environment. The plots above document the agent's journey in meeting the most rewarding performance criteria, where you can see improvements and convergence to higher scores and more consistent decision making.

1. **Conclusion :**

In conclusion, we successfully demonstrated the application of a tabular Q-learning agent in the implementation of a custom Flappy Superhero game. With an agent that learned to survive infinitely, if there was no coin to collect as a win condition, and as well pass all pipes, we carefully designed the environment, reward structure, and training parameters to align with this goal . Random experimentation with many hyperparameters , particularly the epsilon decay rate and learning rate, presented to us just how sensitive RL performance can be to this choice of values.

One key limitation is that the tabular implementation of Q-learning is quite reliant on a discrete and relatively small state space, which may not generalize well to larger, continuous state space environments. Future work could extend this project with Deep Q-Networks (DQNs) suitable for richer state representations, which could provide improved generalization across states. Overall, we have shown the practical ease and difficulties of applying reinforcement learning concepts to a dynamic and interactive game environment.

1. **References :**

Here is all the links to my assets :

GrafxKid. “Classic Hero.” *OpenGameArt.org*, 18 Feb. 2015, opengameart.org/content/classic-hero. Accessed 29 June 2025.

greggman. “Backgrounds for 2D Platformers.” *OpenGameArt.org*, 17 Jan. 2015, opengameart.org/content/backgrounds-for-2d-platformers.

Lexatchison. “2D Coin.” *OpenGameArt.org*, 18 Mar. 2016, opengameart.org/content/2d-coin. Accessed 29 June 2025.