**ENSEEIHT**

# Flappy Bird

Antoine Lescop

Daria Garnier

Med Khoukh

Pierre Durollet

Loïc Capdeville

Contrôle et Apprentissage

2024–2025

# Flappy Bird

In this project, we are simulating a version of the famous Flappy Bird game.

The game features a moving screen where the environment continuously scrolls from right to left.

A single bird is controlled by the player (or the agent). The bird remains horizontally static but can move up or down depending on actions taken.

Pipes regularly appear from the right and move toward the bird.

At each frame, the agent can choose to either jump or do nothing. However, the power of the jump is fixed : we cannot control how strong the jump is.

The difficulty increases every 5 pipes passed, making the game progressively harder as the player advances.

The objective is to survive as long as possible by avoiding collisions with pipes, as well as the top and bottom edges of the screen.

# DNN - Principle

The principle of using a neural model in Flappy Bird is to allow the agent to make decisions based on a set of observable parameters, rather than hard-coded rules.

We model the decision-making process as a weighted sum of key features (like distance to pipes, bird velocity, etc.), and use a simple threshold to decide whether the bird should jump.

This simple strategy can be trained through trial and error using reinforcement learning techniques.

# Reward

We will base the score (and therefore the rewards) on the distance traveled and the number of pipes passed. In special cases later on, we will add bonuses.

# Decision

The decision is simple: JUMP or NOT JUMP.

In our Flappy Bird agent, we use a **weighted sum** of input parameters. If the result is > 0, the bird jumps; otherwise, it doesn't.

This is a basic form of a **linear classifier**, which enables the agent to make decisions based on its current state in the game.

# Learning Methods

## Epsilon greedy

The epsilon-greedy method is a strategy to balance **exploration** and **exploitation**. With a probability ε (epsilon), the agent tries random weights to explore new possibilities. Otherwise, it slightly mutates the current best weights to try to improve upon them.

This helps the agent discover potentially better solutions, especially early in the training.

In practice, we launch a game with a **given set of weights** and evaluate the outcome. If the agent beats the current **highest score**, we save those weights as the new best, which are then used as a basis for future exploitation. For exploration, weights are initialized randomly within the range [−2, 2]. For exploitation, we perturb the best weights slightly by adding a random value between −0.2 and 0.2 to each of them.

We also experimented with normalizing the inputs to keep their magnitudes consistent, but in the end this step turned out to be unnecessary.

## Epsilon Greedy Decaying

This is a variation of the epsilon-greedy strategy where ε (the probability of exploring) decreases gradually over time.

At iteration i, epsilon is updated according to the following formula:

**ε_i = ε_0 × (ε_decay)^i**

**ε_0** is the initial exploration rate (1.0).

**ε_decay** is the decaying factor.

**i** is the current iteration (from 0 to N).

This exponential decay means that the agent starts with a high exploration rate and slowly shifts toward pure exploitation. Since the environment (the game parameters) is stable and doesn't change over time, this method is more efficient in the long run.

We will therefore only keep this strategy in the final training algorithm.

## Greedy simple

This method removes exploration entirely. The agent only performs small mutations around the best-known weights in each generation.

While this can be effective when near an optimal solution, it risks getting stuck in a **local optimum** and missing out on better weight configurations.

Therefore, it is not suited for early training when exploration is crucial.

# Flappy Bird classic

## Reward

SCORE = REWARD = **number_of_pipes** x 1000 + **distance**

## Decision

We have a set of 5 weights, each associated with one of the following : the vertical distance from the top part of the next pipe, the vertical distance from the bottom part of the next pipe, the distance to the next pipe, the falling speed (vertical velocity), and the altitude.

We have carefully selected each input parameters, for example : the altitude is needed to speed up the early learning (not dieing on the edge of the screen), and also later on to not die if the gap between the pipes is too close from the edge of the screen.

```
def should_jump(bird, pipes, weights):
    ...
    decision = (weights[0] * dy_top +
                weights[1] * dy_bottom +
                weights[2] * dx +
                weights[3] * v +
                weights[4] * altitude)

    return decision < 0
```

## Results

The **scores** on the videos are different from the rewards. You can **click on the link** to start the video.

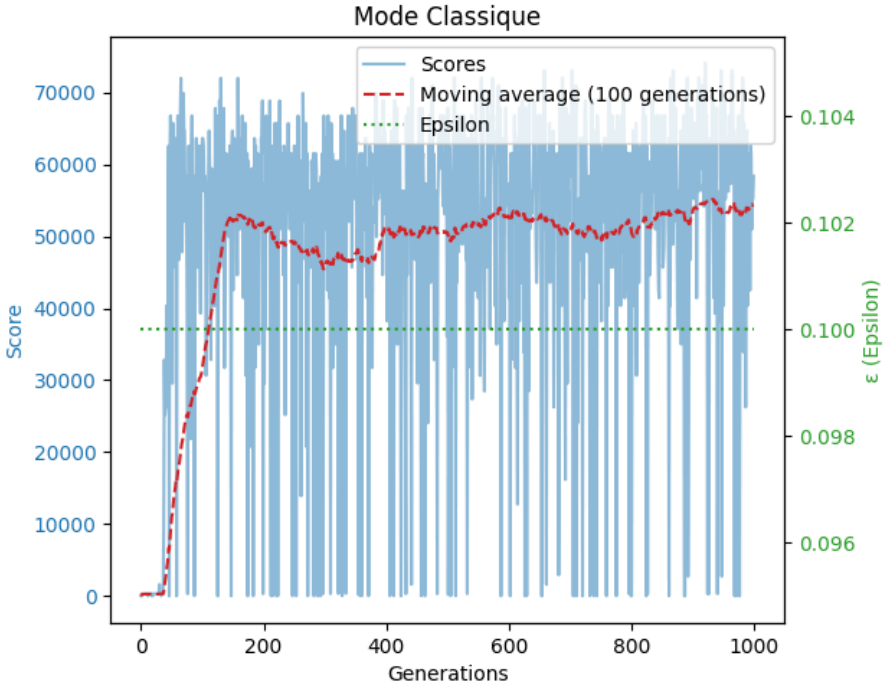
## See video : classic

The agent is getting the perfect score almost every time using the solution weights, he only dies when the game becomes impossible.

### Epsilon greedy

**ε** = 0.1

**number of game** = 1000




)

We observe that the algorithm tends to converge relatively quickly, with the **speed of convergence depending largely on when the first viable solution is encountered**. In the next section, we will repeat the process multiple times and plot the average performance across runs to obtain a clearer view of the learning dynamics.

ε being non null, there is exploitation even at the end of the graph, that's why the results are not consistent even when we already found a good solution to our problem.

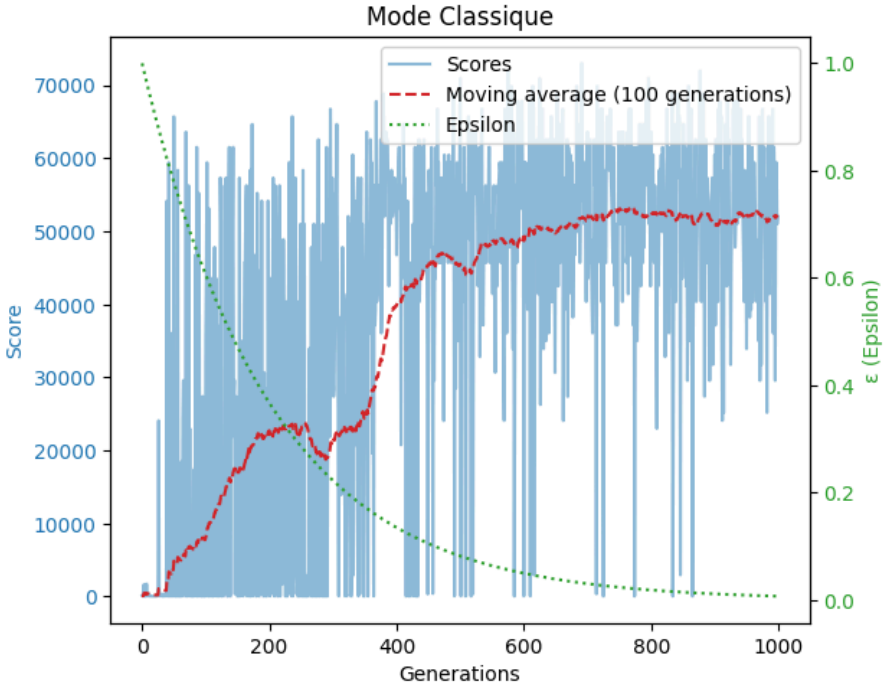
At the end of the simulation (jupyter notebook), we start a game with the best weights we found but we can get unlucky and still get a bad score. We'll try to fix this in the last part.

# Epsilon greedy decaying

**ε_0** = 1

**ε_decay** = 0.995

**number of game** = 1000




)

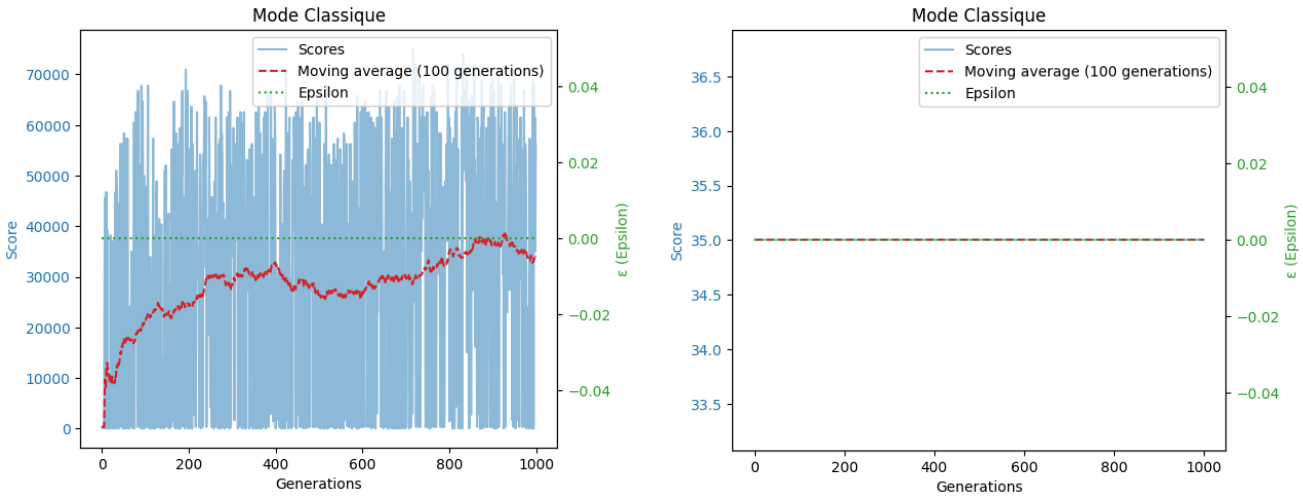
While ε is decaying the moving average is improving because we do more and more exploitation.

At the end ε is close to 0 so we have no exploration, therefor less really bad scores and more consistent results.

## Greedy simple

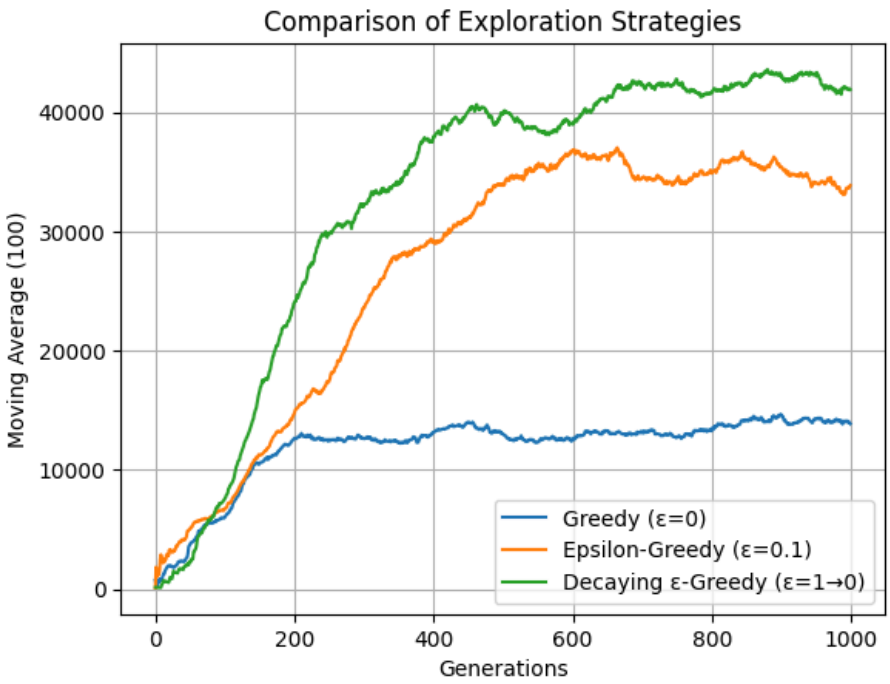
**ε** = 0

**number of game** = 1000




There are 2 possibles issue : the algorithm is stuck in a local maximum (right) or it can start close enough to a good solution and still converge, that's why exploration is mandatory.

## Comparison of the 3 learning methods : Average over 10 training runs




**Greedy simple method < Epsilon greedy method**

Because of the algorithm being stuck in a local maximum sometimes with the greedy simple method, it's not as good as the epsilon greedy method that uses exploration to overcome this issue.

**Epsilon greedy method < Epsilon greedy decaying method**

The decaying epsilon-greedy method is better because it explores more early on to find good solutions quickly, and then reduces exploration later to focus on exploiting the best ones.

PS: We can't make the regret graph because we don't know the optimal/perfect score of each games.

# Flappy Bird complex

## Description

We are adding some features :
- wind
- moving pipes
- unfazed pipes

Expecting not to get perfect score easily.

## Reward

SCORE = REWARD = **number_of_pipes** x 1000 + **distance**

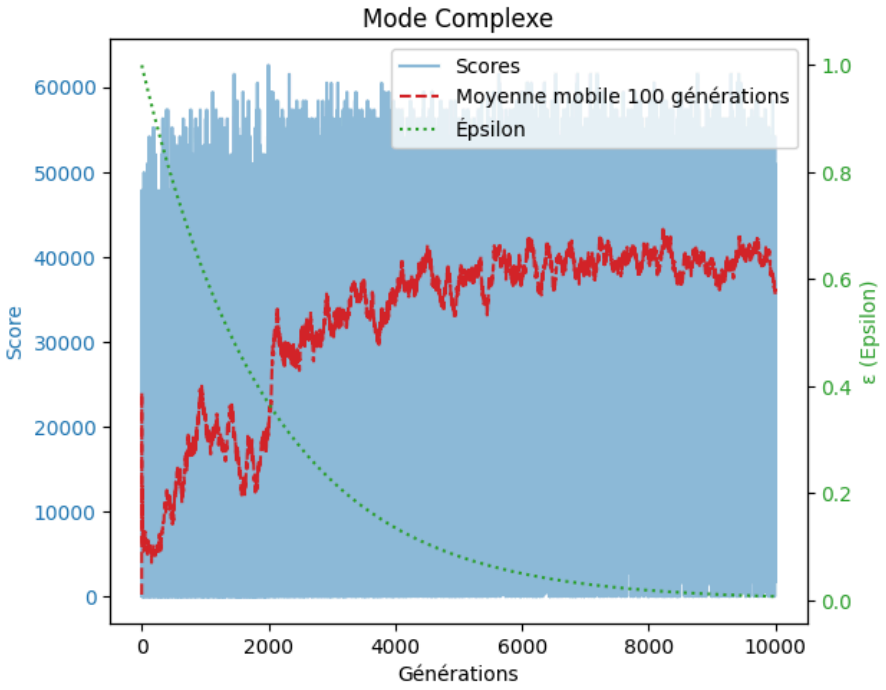
## Decision

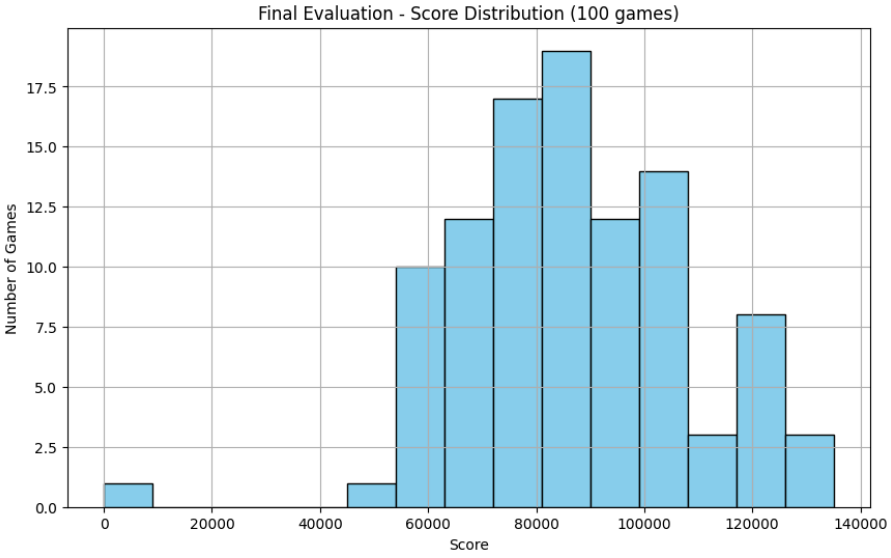
We are adding 3 new inputs : the wind, the pipe's vertical movement and the pipe's horizontal movement. We now have 8 weights to generate.

```
def should_jump_complexe(bird, pipes, weights, wind=0):

    ...

    inputs = [dy_top, dy_bottom, dx, v, altitude, pipe_movement_y, pipe_movement_x, wind]

    decision = sum(w * i for w, i in zip(weights, inputs))
    return decision < 0
```

## Results

## See video : complex




As for the classic version, the solution is close to perfect, the bird almost only dies when it's impossible to pass the pipe.

# Flappy Bird with bonuses

## Description

We are adding some bonus point to gather, when the bird collect them we increase the score (= reward). We hope that the bird will decide to go for them when it's not dangerous and will not die for a single bonus.

(We are going back to the classic Flappy Bird to reduce the number of weights required. This simplification does not significantly impact the learning process.)

# Reward

SCORE = REWARD = 5 x **number_of_bonuses**

We tried a lot of different reward function in order to find a working one, we started off with **number_of_pipes** x 1000 + **distance** + 5000 x **number_of_bonuses** but it doesn't change a lot of things because the best runs are the ones where it flies as far as possible anyway so we remove the distance and pipes part of the reward.

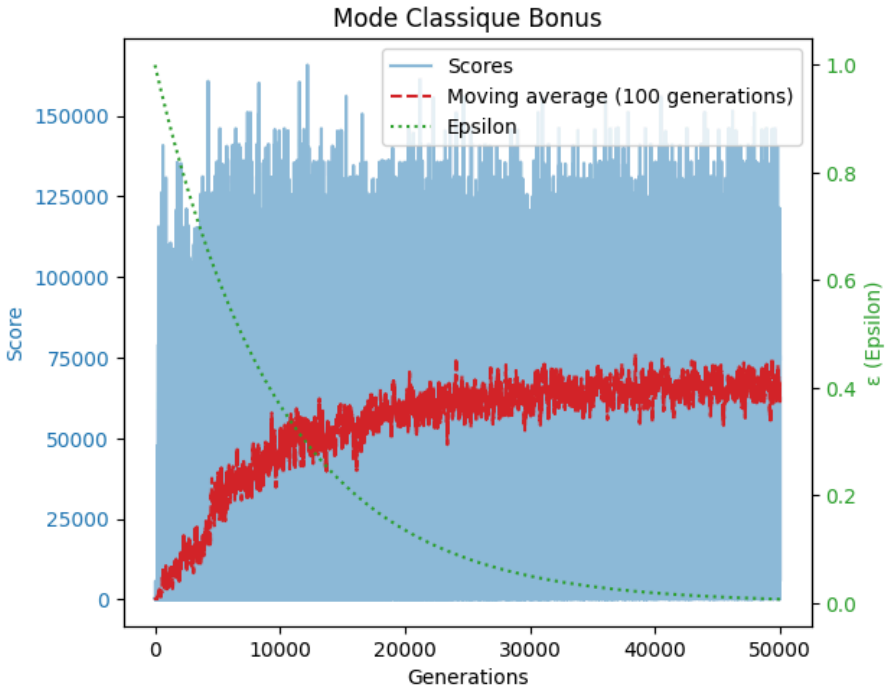
# Decision

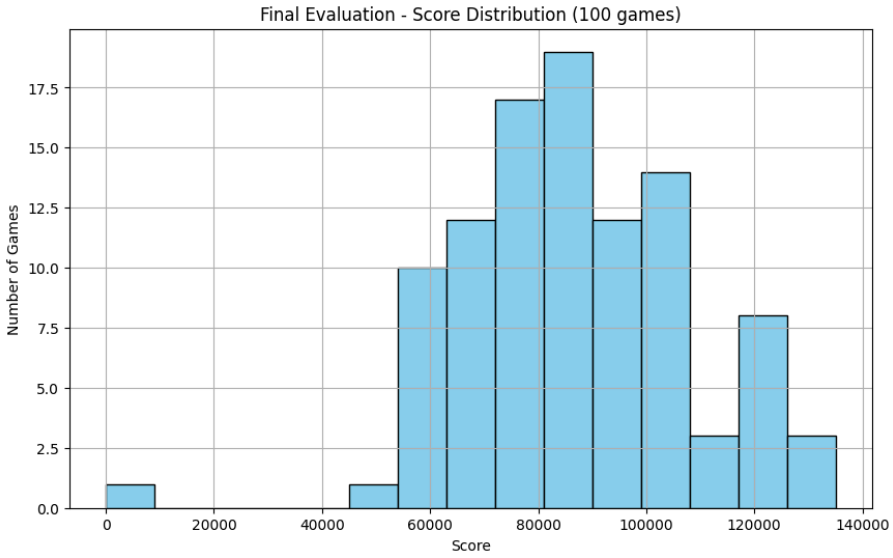
We are adding 2 new inputs : the vertical distance from the next bonus and the horizontal distance from the next bonus.

```
def should_jump_complexe(bird, pipes, weights, bonus):

    ...

    inputs = [dy_top, dy_bottom, dx, v, altitude, dx_bonus, dy_bonus]
    decision = sum(w * i for w, i in zip(weights, inputs))
    return decision < 0
```

# Results

## See video : bonus






✅ Final Weights Evaluation (100 games):

- Average score: 86579.96
- Standard deviation: 21476.47
- Min score: 32
- Max score: 135055

We took the final solution, ran 100 games with it, and plotted a graph to visualize the results.

The graph is not enough to show it but the behavior of the bird is different from the previous experiences, it will sometimes wait before going up or down in order to get some bonuses.

The agent is not perfect because it's not greedy enough, it's not getting all the bonus point he could, but we can expect that a more optimal solution exist but we didn't found it. We need more calculation time and power.

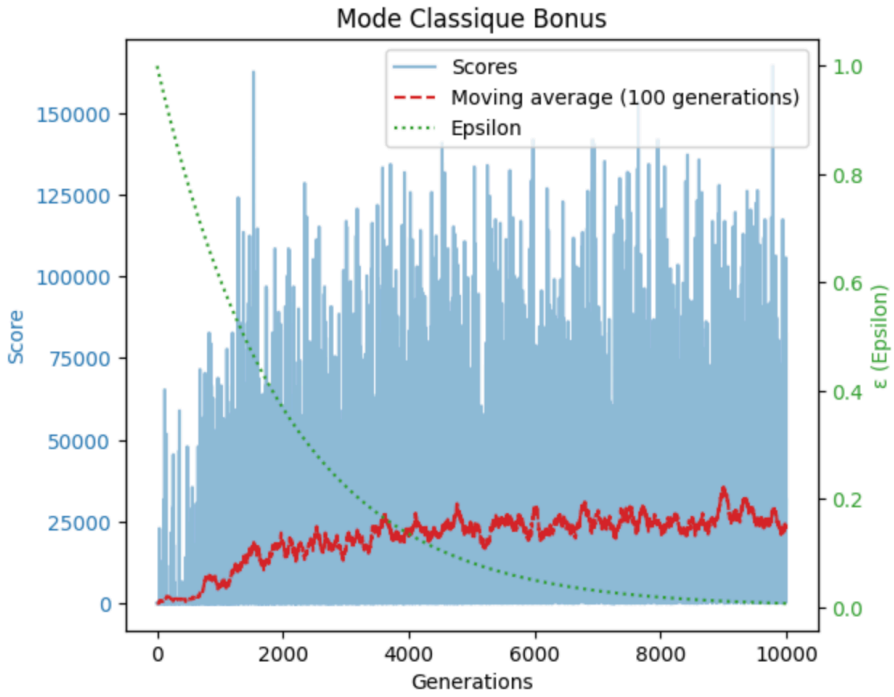
The results are inconsistent (We have 41% of scores below 80 000) not particularly satisfactory, as evidenced by both the graph and the run when executed through the Jupyter Notebook. We'll try to fix the inconsistency in the last part.

# Decision degree 2

One idea to make it more greedy was to add degree 2 inputs. We now have 7 x 2 = 14 weights.

```
def should_jump_complexe(bird, pipes, weights, bonus):

    ...

    inputs = [dy_top, dy_bottom, dx, v, altitude, dx_bonus, dy_bonus]

    inputs_squared = [i**2 for i in inputs]

    full_inputs = inputs + inputs_squared

    decision = sum(w * i for w, i in zip(weights, full_inputs))

    return decision < 0
```

# Results degree 2




The results are worth with the double weights method, but with proper training and enough generations it's possible to get an even better solution with this method.

# Flappy Bird with 2 decisions

## Description

We are keeping the idea of gathering bonuses but we are adding an use to them. The agent can decide to activate the bonus (powerUP) that is boosting the score by 2 ways, multiplying the score while using the powerUP and by surviving until the end of the bonus. We played a lot with these reward parameters in order to push the agent toward these 2 objectives :

**Objective n°1 :** Gather powerUPs (same as last game)

**Objective n°2 :** Use powerUP at the right timing (just before passing a pipe)

However, achieving a perfect result is challenging. We will explain this later, but we did not include all the tests of reward calculation formulas in order to avoid overloading the report.

## Reward

REWARD = **number_of_pipes** x 1000 + **distance** + **number_of_powerUp_used_and_survived** x 1000 + **number_of_pipes_while_using_powerUP** x 100000

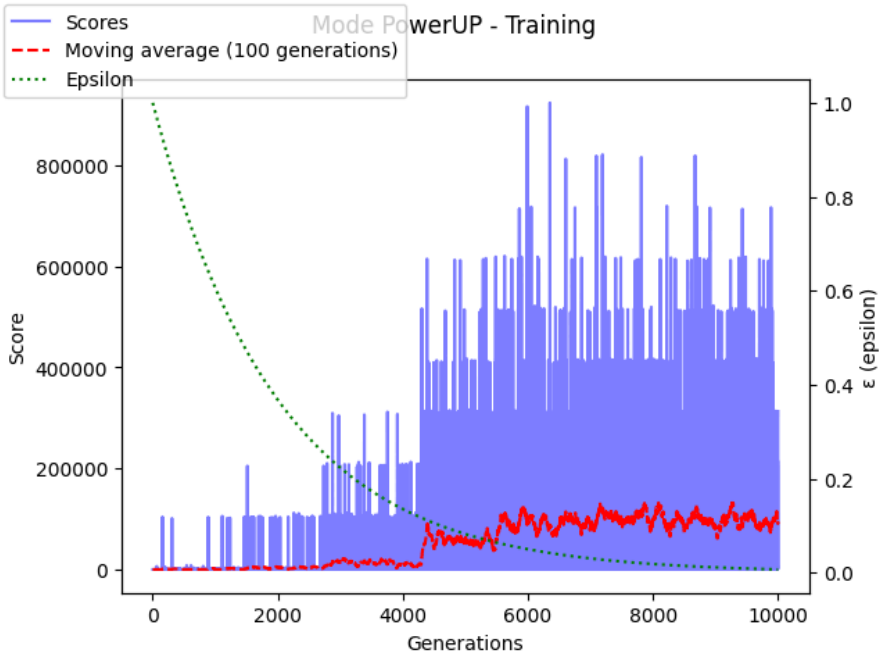
## Decision

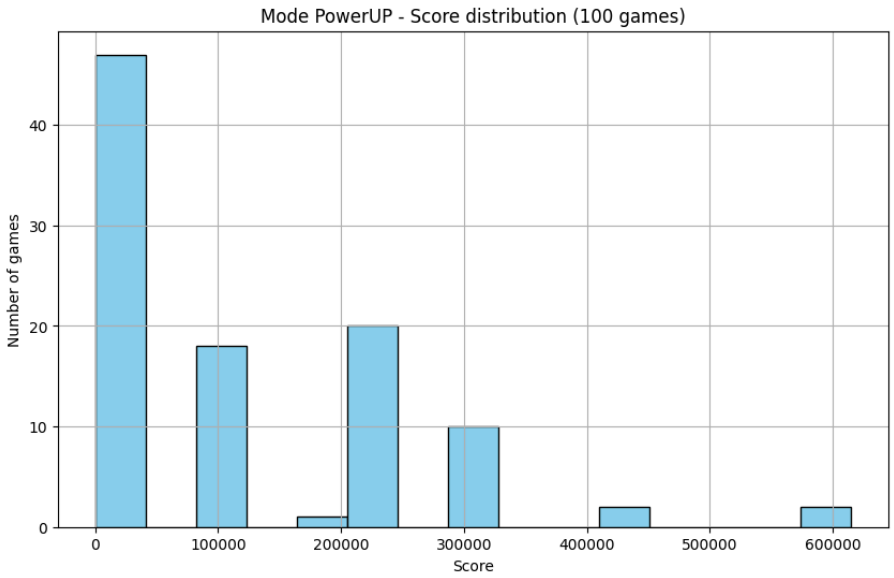
We have 2 sets of 7 weights, one set for each decision.

```python
def should_jump_complexe(bird, pipes, weights_jump, weights_powerup, wind=0, powerups=[]):

    ...

    jump_inputs = [
        bird.y - next_pipe.height,
        bird.y - (next_pipe.height + PIPE_GAP),
        next_pipe.x - bird.x,
        bird.velocity,
        bird.y,
        dx_powerup,
        dy_powerup
    ]

    power_up_inputs = [
        bird.y - next_pipe.height,
        bird.y - (next_pipe.height + PIPE_GAP),
        next_pipe.x - bird.x,
        bird.velocity,
        bird.y,
        dx_powerup,
        dy_powerup
    ]

    jump_decision_value = sum(w * i for w, i in zip(weights_jump, jump_inputs))
    powerup_decision_value = sum(w * i for w, i in zip(weights_powerup, power_up_inputs))

    jump = jump_decision_value < 0
    use_powerup = powerup_decision_value < 0

    return jump, use_powerup
```

# Results

## SEE VIDEO : powerUP






**Objective 1 :** This version gather powerUPs less successfully than the previous one, as it involves twice as many coefficients to optimize, making it harder to achieve an optimal result. However, I remain convinced that with enough iterations, it is possible to approach a more ideal solution.

**Objective 2 :** Regarding the use of the powerUP at the right timing (i.e., just before passing a pipe), we encountered a significant challenge, our agent is often using it as soon as possible: The first decent solutions emerged when the bird traveled far enough to trigger the use of a powerUP—meaning it would activate the powerUP immediately after collecting it. In these cases, we fix all coefficients to these values as a starting point for further exploration.

However, this introduced a flaw: we should have only fixed the jump-related coefficients while leaving the powerUP activation coefficients free to evolve. By freezing the entire set, we limited the agent's ability to improve its decision-making for using the powerUP at the most strategic moments.

We tried :

- using REWARD = **number_of_pipes_while_using_powerUP**
- giving him some powerUP to use at the beginning, hoping that it will learn when to use powerUPs but instead it still learned how to go as far as possible and still use powerUPs as soon as possible.

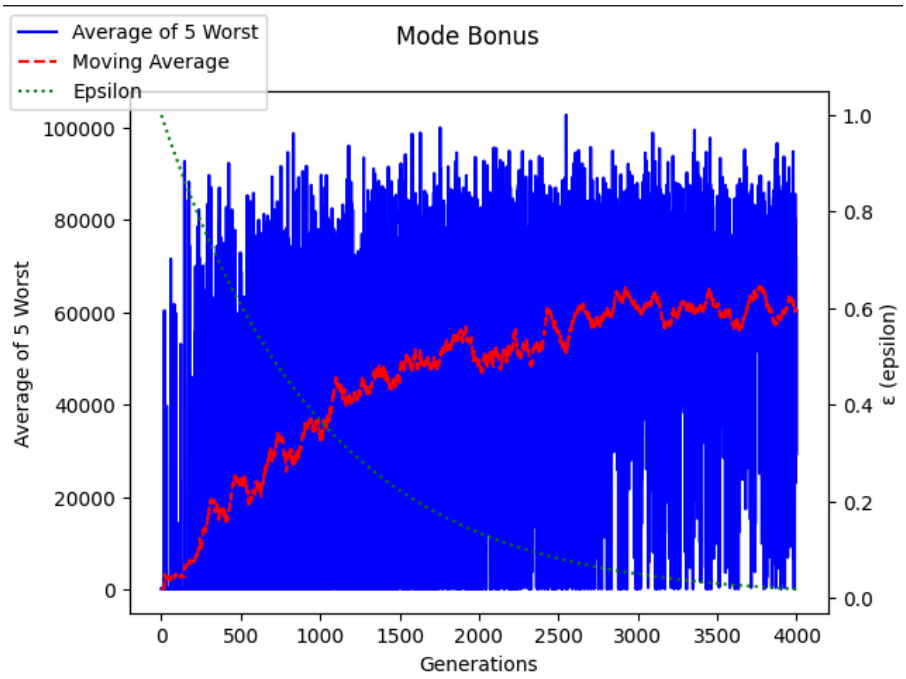
# Flappy Bird learning using smart batches (Bonus point version)

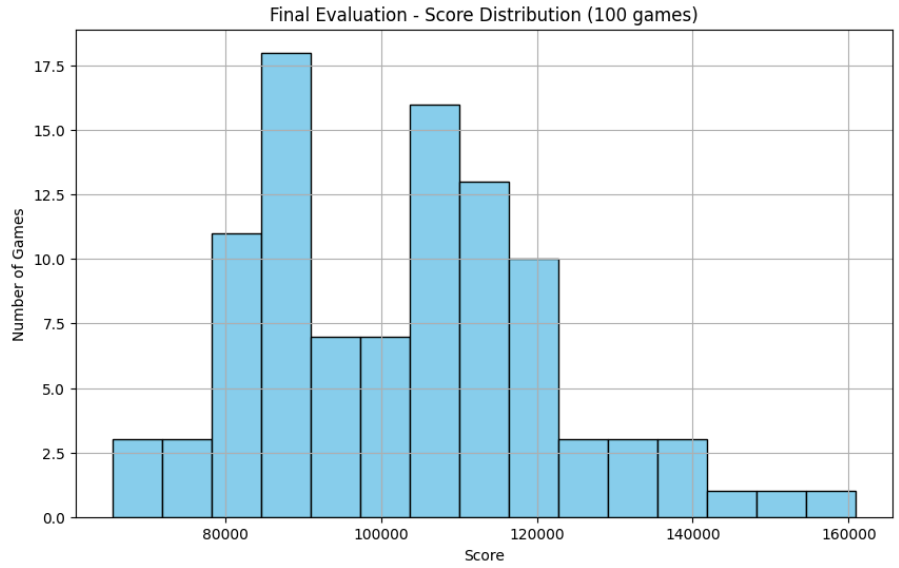
In previous approaches, the weights were selected based on achieving the highest score. However, this does not necessarily reflect the most reliable or consistent solution. A single high-scoring episode might be the result of randomness or lucky decisions, rather than a robust strategy. This can lead to overfitting to rare successful cases rather than learning a generalizable behavior. That's why, when running the special versions of the game in Jupyter Notebook, we sometimes observe very poor scores during the rendered gameplay.

By using smart batch selection, we aim to improve learning stability by considering the overall consistency of performance across multiple episodes.

# Using the average of the 5 worst results out of 10 games

Using this method we are trying to avoid overfitting to a lucky score. The solution is more reliable and we got less very bad game using it.




✅ Final Weights Evaluation (100 games):

- Average score: 103112.88
- Standard deviation: 19054.12
- Min score: 65617
- Max score: 160864

Without using this method (see the “Flappy Bird with bonuses” part) we had 41% of games below a score of 80 000, we now have only 6% of games below this threshold. Also the average score as increased from 86 000 to 103 000.

# Using standard deviation to measure consistency

While using the average of the 5 worst scores helps prevent overfitting to lucky runs, we can go even further by evaluating how consistent a batch is. This is where the standard deviation comes in.

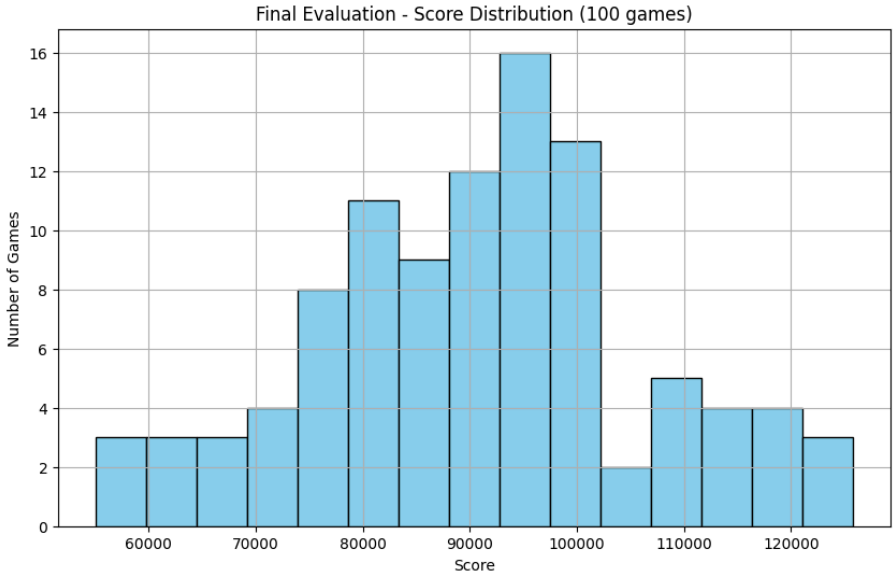
The standard deviation measures how spread out the scores are. A lower standard deviation means the agent performs similarly across games : it's more stable. A high average score with a low standard deviation indicates both strong and consistent performance.

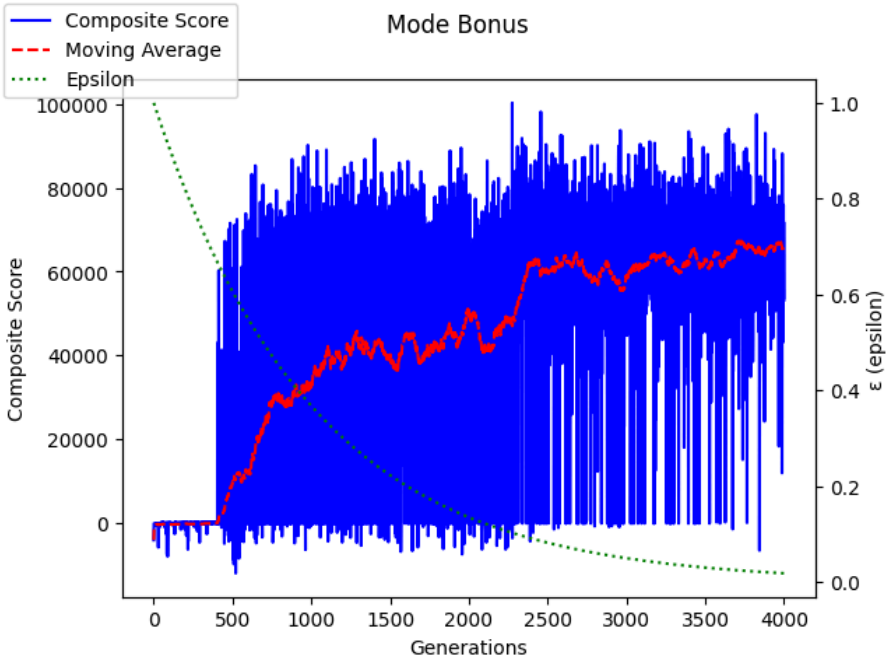
We can test a new method that selects the best weights not just based on the average of the 5 worst scores, but based on a composite score like this:

composite_score = average_of_worst_5 - (k × standard_deviation)

Here, k is a tunable parameter to penalize inconsistency. The goal is to maximize performance while minimizing variance.

This approach favors agents that not only play well but do so reliably.



✅ Final Weights Evaluation (100 games):

- Average score: 91083.02
- Standard deviation: 16419.03
- Min score: 55069
- Max score: 125750

With this method, we aim to reduce the standard deviation of the results, making the agent's performance more consistent across games. In our tests, we successfully lowered the standard deviation by about 2,600 points, though this came at the cost of a slightly lower average score. However, this kind of comparison is not yet fully rigorous: to properly evaluate both strategies, we would need to repeat the experiment many times for each method. This would allow us to clearly show that we can target and optimize for specific goals, such as consistency or peak performance. Unfortunately, doing so would require significantly more time than is currently feasible.