

# Agents

Mário Antunes

February 11, 2026

## Practice Guide: Flappy Bird with Neuro-evolution

In this guide, you will transition from a basic reflex agent to a sophisticated AI capable of learning how to play Flappy Bird using **Neuro-evolution**.

The game architecture is split into two parts:

1. **Backend (Python):** Runs the game physics and logic using asynchronous websockets.
2. **Frontend (HTML5/Canvas):** Visualizes the state sent by the server.

### Part 1: Setup

Before writing any code, let's get the game running.

1. **Download the Repository:** Clone the course repository:

```
git clone https://github.com/detiuaveiro/flappy-bird-agent.git
cd flappy-bird-agent
```

2. **Create a Virtual Environment:** It is best practice to isolate your dependencies.

```
python3 -m venv venv
source venv/bin/activate
```

3. **Install Dependencies:** Install the required libraries.

```
pip install -r requirements.txt
```

4. **Run the Game:** Start the backend server:

```
python -m src.backend.py --pipes
```

Once the server is running, open the `html/play_human.html` file in your web browser. Use the mouse to play the game.

## Part 2: The Basic Agent (Reflex Agent)

Your first task is to understand the communication protocol. The server sends the **World State** as a JSON object via websockets.

**Task:** 1. Review `play_game` method in `play.py`. 2. Locate the message handling loop. You will see a JSON object representing the state. 3. Implement a simple **Rule-Based Agent**. Do not use Machine Learning yet. Write a simple `if` statement.

**Example Logic:**

```
# A simple reflex agent
if state['bird_y'] > state['next_pipe_bottom_y'] + 50:
    action = "jump"
else:
    action = "stay"
```

Run this agent and observe. Does it crash? Can it handle different pipe heights?

## Part 3: The Intelligent Agent (Neuro-evolution)

Now we move to the core of this course: **Neuro-evolution**.

### 1. Theoretical Background

Why use Neuro-evolution for Flappy Bird?

- **Continuous World:** The bird's position, velocity, and pipe locations are continuous values (floats).
- **Discrete Action:** The output is binary (Jump or Don't Jump).
- **Partial Observability:** The agent doesn't need to know the entire map; it only needs a "partial view" (distance to the next pipes).

### The Solution: Neural Networks + Evolutionary Algorithms

Instead of using Reinforcement Learning (like Q-Learning) which requires a state table (hard for continuous worlds) or Gradient Descent (which requires a dataset of "correct" moves), we use **Neuro-evolution**.

- **The Brain (MLP):** We use a Multi-Layer Perceptron.
- **Input:** Normalized state (Distance to pipe X, Distance to pipe Y, Velocity).
- **Hidden Layers:** Processing units.

- **Output:** A single neuron with a Sigmoid activation. If Output > 0.5 Jump.
  - **The Optimization (Evolution):** We do not “train” the network with back-propagation. Instead, we **evolve** the weights of the neural network.
1. **Population:** Generate 50 birds with random weights.
  2. **Evaluation:** Let them all play. The “Fitness” is the score (distance traveled).
  3. **Selection:** Keep the weights of the birds that flew the furthest.
  4. **Crossover & Mutation:** Create a new generation of birds by mixing the weights of the best parents and adding random mutations.

## 2. Implementation

Open the `train.py` file. The repository is set up to use a Python optimization library ([PyBlindOpt](#); take some time to browse the [documentation](#)).

**Your Task:** Configure the Neuro-evolution loop.

1. **Define the Problem:** Map the Neural Network weights to a 1D vector. The optimization algorithm will try to find the vector of weights that maximizes the game score.
2. **Select an Algorithm:** The library supports several optimization algorithms, such as:
  - **DE (Differential Evolution):** Good for global search, maintains diversity.
  - **PSO (Particle Swarm Optimization):** Simulates a flock of birds finding food.
  - **GWO (Grey Wolf Optimizer):** Mimics the leadership hierarchy of wolves.
  - **CS (Cuckoo Search):** Based on the brood parasitism of cuckoo birds.

## 3. Experiments

Run the training using different configurations and observe the impact on convergence speed (how fast they learn) and stability (do they forget how to fly?).

**Experiment A: Algorithm Comparison** Run the training for 30 generations with a population of 30 for each algorithm:

- Does **PSO** converge faster than **DE**?
- Does **GWO** find a more stable solution?

**Experiment B: Initialization** Check the documentation for the optimization library regarding [Initialization Metrics](#).

- How does the range of initial random weights (e.g.,  $[-1, 1]$  vs  $[-5, 5]$ ) affect the learning?
- If weights start too large, the sigmoid output might saturate (always 1 or always 0), making the bird constantly jump or never jump.

## Part 4: The Flappy Bird Cup

**Friendly Competition:** Once you have explored the algorithms, tune your hyperparameters to create an optimal agent.

- **Input features:** Are you feeding the agent `bird_y`? Or `bird_y - pipe_y` (relative distance)? (Hint: Relative distance usually generalizes better).
- **Hidden Layers:** Is a standard [3 inputs -> 5 hidden -> 1 output] enough? Or do you need two hidden layers?
- **Population Size:** Larger populations explore better but run slower.

**Goal:** Train an agent that can pass **100 pipes** consistently.