Hayden Wood

Andre Ellis

**AI Agent Flappy Bird Report**

## **Introduction**

In this project, we worked on creating a framework for training an AI agent to play Flappy Bird, focusing on the methodology, challenges, and potential outcomes. While we didn’t write any code or run experiments, our approach outlines key considerations and solutions for implementing RL in game environments.

## **Approach**

### **Environment Setup:**

The Flappy Bird environment revolves around simple physics: the bird falls due to gravity and flaps upward when prompted. The AI interacts with this environment by observing the game’s state (e.g., the position of the bird and pipes) and choosing actions (flap or not).

#### **Tools and Libraries:**

We would use:

* PyGame to recreate or modify the game environment, as it’s beginner-friendly and flexible for 2D games.
* OpenAI Gym for standardizing the interaction between the AI and the environment. If a Flappy Bird environment compatible with Gym already exists, we’d use that for simplicity.

#### **State Representation:**

The state would consist of the game’s visual frames, resized to 84×84 pixels to reduce computational overhead. To make it easier for the AI to understand the game dynamics, we’d convert the frames to grayscale and stack the last four frames together. This way, the AI can see motion and understand the bird’s velocity and position relative to the pipes.

#### **Action Space:**

The action space would include only two options:

* 0: Do nothing, allowing the bird to fall.
* 1: Flap, causing the bird to move upward.

#### **Reward System:**

Rewards would guide the AI’s learning:

* +1 for passing a pipe.
* -1 for collisions with pipes or the ground.
* A small negative reward per frame (-0.01) to encourage the AI to complete the game more efficiently.

### **Pre-trained Model Integration**

To help the AI process game frames, we’d use transfer learning. A pre-trained model like MobileNetV2 or ResNet could be leveraged to extract visual features from the game.

#### **Modifications:**

We’d remove the final classification layers of the pre-trained model and replace them with layers designed for reinforcement learning. These layers would feed into a Q-network to predict action values based on the game’s current state.

#### **Challenges:**

One major challenge is that pre-trained models are designed for real-world images, not game frames. To address this, we’d preprocess the frames to emphasize important features, like edges and contrasts, which align better with the data these models were trained on.

### **Reinforcement Learning Implementation**

We decided that Deep Q-Learning (DQN) would be the best RL algorithm for this project. DQN is effective for problems with continuous state spaces (like game frames) and discrete action spaces (flap or no flap).

#### **Key Components:**

* Q-Network: A convolutional neural network that takes the game frames as input and predicts Q-values for each action.
* Replay Memory: Stores the agent’s past experiences (state, action, reward, next state) to break the correlation between consecutive actions during training.
* Target Network: A secondary Q-network that is updated less frequently to stabilize learning and prevent the AI from chasing moving Q-value targets.

#### **Exploration-Exploitation Trade-off:**

We’d use an ε-greedy policy, where the agent starts with a high ε (exploration) and gradually reduces it over time to focus more on exploitation (choosing the best-known action).

#### **Experience Replay:**

To make training more stable, we’d sample mini-batches of experiences from replay memory rather than using consecutive frames. This helps the agent learn more general patterns instead of overfitting to specific sequences.

### **Training and Evaluation**

#### **Training Process:**

The training process would look something like this:

1. Initialize the environment and the Q-network.
2. Let the agent interact with the environment, choosing actions based on its policy.
3. Record experiences (state, action, reward, next state) and store them in replay memory.
4. Periodically train the Q-network using random samples from replay memory.
5. Update the target network every few iterations to stabilize learning.

#### **Evaluation Metrics:**

We’d evaluate the agent using two key metrics:

* Average Score: How many pipes the bird passes, on average, per episode.
* Survival Time: How long the bird stays alive during each episode.

## **Challenges and Solutions**

### **Challenge 1: Sparse Rewards**

The agent only gets rewards when passing pipes, which might make learning slow. To address this, we’d add small negative rewards each frame, encouraging the agent to stay alive and make progress.

### **Challenge 2: Stability of Training**

Sequential data can make training unstable. We’d solve this by using replay memory and a target network, ensuring that the agent learns from a more stable and diverse set of experiences.

### **Challenge 3: Domain Adaptation**

Pre-trained models are trained on real-world images, which don’t resemble game frames. To bridge this gap, we’d preprocess the frames (e.g., grayscale conversion, edge detection) to emphasize the most important features for decision-making.

### **Challenge 4: Balancing Exploration and Exploitation**

Striking the right balance between trying new actions and sticking with what works is tough. Using an ε-greedy policy with a gradual decay of ε would ensure the agent explores more at the start and exploits its knowledge as it improves.

## **Results and Analysis**

Although we didn’t implement the project, here’s what we’d expect:

* Early in training, the agent would behave randomly and crash quickly.
* Over time, as it learns, it would survive longer and pass more pipes consistently.
* Performance graphs (e.g., average score vs. episodes) would show a steady upward trend as the agent masters the game.

We’d also visualize the Q-values to see how the agent’s decision-making improves. Comparing the AI’s performance to a random action baseline and human benchmarks would help us gauge its success.

## **Reflections**

This project was a great opportunity to dive deeper into reinforcement learning and computer vision. We learned how RL systems depend on well-designed rewards and stable training methods. It was fascinating to see how every part—like preprocessing, the Q-network, and replay memory—plays a role in the agent’s success.

One thing that stood out to us was how tweaking something as small as a reward function or frame stack could significantly impact the AI’s performance. It also reinforced how much trial and error is involved in designing RL systems.

## **Potential Improvements and Future Work**

### **Improvements:**

* Curiosity-Driven Exploration: Adding intrinsic rewards for exploration could help the agent learn more effectively in sparse-reward environments.
* Recurrent Models: Using RNNs or Transformers to process sequential data might improve the agent’s ability to anticipate future states.
* Dynamic Difficulty: Gradually increasing game speed or pipe frequency could create a more robust and adaptable agent.

### **Future Work:**

* Testing the framework on more complex games with larger action spaces.
* Extending the project to multi-agent systems, where multiple birds cooperate or compete.
* Experimenting with other RL algorithms, like Proximal Policy Optimization (PPO), to see if they yield better results.

## **Conclusion**

This project helped us understand the integration of computer vision and reinforcement learning to train an AI agent. By breaking down the process step-by-step, we gained insights into how to set up environments, design reward systems, and stabilize training. While conceptual, our approach highlights the challenges and potential of using RL in game environments. With further refinement and implementation, this framework could lead to exciting real-world applications beyond games.