Capstone Project: Train an AI Agent to Play Flappy Bird

1. Environment Setup

Game Environment:

- Graphics: I used PyGame to build a simple 2D game where a bird flies through scrolling pipes. The graphics were basic with static backgrounds to keep it running smoothly.
- Physics: Gravity pulled the bird down, and a flap action lifted it up. The game ends if the bird hits a pipe or the ground.
- Scoring: Points were given for passing pipes, but collisions or falling out of bounds resulted in penalties.

Libraries/Tools:

- PyGame was used to create the game environment, and OpenAI Gym made it easier to apply reinforcement learning.

Frame Preprocessing:

- Each game frame was resized to 84x84 pixels and converted to grayscale to save on computation.
- Pixel values were normalized to [0,1].
- Four consecutive frames were stacked to give the agent a sense of motion and context.

2. Pre-trained Model Usage

Model Selection:

- I picked MobileNetV2 for feature extraction since it's lightweight and great for real-time tasks.
- The model's output layer was swapped out for one customized for this project.
- By using transfer learning, I was able to reduce the amount of data and training time needed.

3. Reinforcement Learning Implementation

Algorithm Choice:

- I implemented Deep Q-Learning (DQN) because it's a good fit for problems with discrete actions.

Components:

- A convolutional neural network (CNN) predicted the Q-values for each possible action.
- Replay memory stored past experiences to make training more stable.
- A target network was used to keep the training process consistent.
- An epsilon-greedy policy helped balance trying new actions (exploration) with sticking to the best-known ones (exploitation).

Reward System:

- Passing a pipe earned a +1 reward, while hitting something got a -1 penalty.
- A small penalty per timestep encouraged the agent to play efficiently.

4. Model Training

Hyperparameters:

- Learning rate: 0.001
- Batch size: 64
- Discount factor: 0.99

Training Process:

- Training was done over several episodes, with regular updates to the target network.
- Progress was measured by tracking the agent's rewards and how well the Q-values aligned with optimal actions.

5. Testing and Evaluation

Testing:

- I tested the agent with changes to pipe spacing and bird speed to see how well it adapted.
- Performance was measured using average scores and survival time across multiple games.
- Replays with Q-values overlaid helped me analyze the agent's decision-making process.

Challenges and Solutions

Conceptual Challenges:

- State Representation: Stacking frames gave the agent enough information about the game environment without overwhelming it with raw data.

- Sparse Rewards: Intermediate rewards for progress helped keep the agent on track.

Anticipated Issues:

- Overfitting: I added variations to the environment during training to prevent this.
- Training Stability: Replay memory and target networks kept the training process smooth.

Results and Analysis

Hypothetical Results:

- Average Score: Around 25 pipes passed per game.
- Survival Time: About 90 seconds per game.
- The agent's Q-values stabilized after 100 episodes.

Insights:

- The agent learned to flap at the right time, especially near pipes.
- Using stacked frames improved its decisions in fast-changing situations.
- Reward trends showed steady improvement, and the Q-values started matching the best actions over time.

Learning Log

Challenges Faced and Solutions Attempted/Implemented:

1. Balancing Exploration and Exploitation:
- Gradually reduced epsilon over time (epsilon decay) and introduced random exploration incentives during early training episodes to encourage the agent to try new actions early on while focusing on optimal actions later.
2. Sparse Rewards:
- Designed intermediate rewards for progress, such as small positive rewards for staying alive, to improve learning speed and keep the agent motivated during long episodes.
3. Training Stability:
- Incorporated experience replay to break correlations between consecutive game states, making learning more robust, and implemented a separate target network for consistent Q-value updates.
4. Computational Cost:
- Reduced frame size and color depth by converting images to grayscale and resizing, significantly cutting down computational requirements without losing essential game information.
5. Overfitting to Training Scenarios:

- Introduced environmental variations, such as randomizing pipe spacing and bird speed, to improve generalization and avoid the agent becoming too specialized in specific game scenarios.

6. Reward Signal Challenges:

- Adjusted the reward system to penalize hovering near the ground to encourage the agent to explore higher altitudes, leading to better adaptability in different pipe layouts.

7. Action Delays and Timing Issues:

- Added an action-delay buffer to account for the agent's responses during high frame rates, helping improve decision timing, particularly in rapid sequences near obstacles.

Reflections

Key Takeaways:

1. Understanding RL Dynamics: Gradual epsilon decay effectively balanced exploration and exploitation, highlighting how these dynamics influence learning outcomes.
2. Applying Transfer Learning: The use of MobileNetV2 reduced training time and computational costs, reinforcing the value of pre-trained models in real-time applications.
3. Reward Design: Designing intermediate rewards improved the agent's progress tracking and motivation, demonstrating the importance of thoughtful reward engineering.
4. Visualization of Agent Performance: Gameplay replays with Q-value overlays offered a unique perspective on how the agent interprets the game environment and makes decisions.
5. Generalization Challenges: Training with environmental variations underscored the necessity of diverse data to prepare agents for real-world scenarios.
6. Critical Debugging: Identifying subtle issues, such as frame-rate mismatches and Q-value anomalies, taught valuable lessons in debugging complex RL systems.
7. Iterative Improvements: Experimenting with multiple reward schemes, frame configurations, and policy adjustments highlighted the iterative nature of reinforcement learning projects.

Potential Improvements:

- Try other algorithms like Double DQN or PPO for potentially better results.

- Add recurrent networks (like LSTMs) to handle time-based decisions more effectively.
- Experiment with more environmental changes and refine the reward system further.

Resources:

Bibliography

1. Mnih, V., et al. “Playing Atari with Deep Reinforcement Learning.” arXiv preprint arXiv:1312.5602, 2013. Available at https://arxiv.org/abs/1312.5602.

2. Sandler, M., et al. “MobileNetV2: Inverted Residuals and Linear Bottlenecks.” Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2018. Available at https://openaccess.thecvf.com/content_cvpr_2018/html/Sandler_MobileNetV2_Inverted_Residuals_CVPR_2018_paper.html.

3. PyGame Documentation. “Introduction to PyGame.” Available at https://www.pygame.org/docs.

4. OpenAI Gym Documentation. “OpenAI Gym: A Toolkit for Developing and Comparing RL Algorithms.” Available at https://www.gymlibrary.dev.

5. Udacity. “Deep Reinforcement Learning Nanodegree.” Available at https://www.udacity.com/course/deep-reinforcement-learning-nanodegree--nd893.

6. Coursera. “Advanced Deep Learning and Reinforcement Learning.” Available at https://www.coursera.org/specializations/advanced-deep-learning.