Capstone Project: Conceptual Path – Training an AI Agent to Play Flappy Bird

# 1. Environment Setup

Flappy Bird is a side-scrolling game where the player controls a bird attempting to fly between columns of green pipes. The game environment is composed of simple 2D graphics, basic physics for gravity and velocity, and a scoring system based on how many pipes the bird successfully passes through.  
  
To simulate this environment for AI interaction, I would use the PyGame Learning Environment (PLE), which is a wrapper around PyGame games for reinforcement learning experiments. Additionally, I’d use OpenAI Gym for environment interaction patterns, and OpenCV for image preprocessing.  
  
State representation would include the bird’s vertical position, velocity, and the positions of the pipes. The action space would consist of two actions: flap (upward movement) and no action (gravity pulls the bird down). The reward system would be: +1 for passing a pipe, -100 for crashing, and 0 for other frames.  
  
Frames would be preprocessed by converting them to grayscale and resizing them (e.g., to 84x84 pixels), to reduce input size and highlight essential features.

# 2. Pre-trained Model Usage

Transfer learning involves using a model trained on one task to assist with another related task. This is useful for saving training time and improving performance with limited data.  
  
I would use MobileNetV2, a lightweight convolutional neural network pretrained on ImageNet, ideal for feature extraction in resource-limited settings. To adapt MobileNetV2, I’d remove the classification head and use the convolutional base to extract spatial features from Flappy Bird frames.  
  
Challenges include the domain gap between real-world images (ImageNet) and game environments. To overcome this, I’d fine-tune the final convolutional layers or train additional dense layers on top to specialize in game-specific patterns.

# 3. Reinforcement Learning Implementation

Reinforcement learning (RL) involves an agent interacting with an environment to learn optimal actions by maximizing cumulative rewards. Key concepts include:  
- States: Game frames  
- Actions: Flap or no flap  
- Rewards: Game-defined scores  
- Policy: A strategy to select actions  
  
I would implement Deep Q-Learning (DQN), which uses a neural network (Q-network) to approximate Q-values for each action. The architecture would include convolutional layers for feature extraction, followed by dense layers to estimate Q-values.  
  
Replay memory stores experiences (state, action, reward, next\_state), and a target network is used for stable Q-value estimation.  
  
To balance exploration and exploitation, I’d use an epsilon-greedy strategy, where the agent chooses random actions with probability epsilon, which decays over time. Experience replay improves training stability by breaking correlations between sequential experiences.

# 4. Model Training

The training loop would involve the agent interacting with the environment, storing experiences, and updating the Q-network using mini-batches from the replay memory.  
  
Steps include:  
- Initialize environment and Q-network  
- Observe initial state  
- Repeat: choose action, execute, observe next state/reward, store in memory  
- Sample batch and update Q-network  
  
Important hyperparameters: learning rate, discount factor (gamma), batch size, epsilon decay, target update frequency.  
  
Training issues like reward sparsity (few meaningful rewards) could be mitigated by shaping rewards or early training on simpler versions of the game.  
  
Model performance would be evaluated using average score and consistency across episodes.

# 5. Testing and Evaluation

Testing the agent would involve running it in the environment without learning (no training updates), to measure how well it plays.  
  
Metrics:  
- Average score across multiple episodes  
- Survival time (number of frames before game over)  
  
Performance would be visualized using graphs (e.g., score per episode) and by recording gameplay.  
  
Results would be benchmarked against random agents and early versions of the model.  
  
Improvements could include trying other RL algorithms (e.g., Double DQN, PPO), enhancing the reward system, or using attention mechanisms to focus on relevant features.