Flappy Bird AI Reinforcement Learning Project

This notebook implements a Deep Q-Network (DQN) to learn how to play the Flappy Bird game using a pre-trained MobileNetV2 model as a feature extractor.

High-Level Steps:

1. Environment Setup
2. Pre-trained Model Integration
3. Reinforcement Learning Implementation (DQN)
4. Model Training
5. Testing and Evaluation

Throughout each step, I have provided explanation what the section is performing and why to ensure the clarity.

1. Environment Setup

I began by importing all the necessary libraries and ensuring reproducibility by setting random seeds. To interact with the Flappy Bird environment, I relied on the PyGame Learning Environment (PLE), configuring it so that no screen is displayed, which made training faster. I then defined a `preprocess` function to resize and normalize the frames, preparing them for MobileNetV2 input. Given past attempts, I decided to provide a substantial survival reward (+0.1) in `get_reward`, hoping that this stronger incentive would encourage the agent to survive longer. For collision detection, I kept it simple with a rectangle-based check to maintain efficiency.

In [1]:

```python
import numpy as np
import cv2
import matplotlib.pyplot as plt
from collections import deque
import random
from tqdm import tqdm
import os
import pygame

np.random.seed(42)
random.seed(42)

from ple.games.flappybird import FlappyBird
from ple import PLE

import tensorflow as tf
from tensorflow.keras.applications import MobileNetV2
from tensorflow.keras.layers import GlobalAveragePooling2D, Dense
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import Adam

print("TensorFlow version:", tf.__version__)
print("Num GPUs Available:", len(tf.config.list_physical_devices('GPU')))

game = FlappyBird()
env = PLE(game, fps=30, display_screen=False)
env.init()
action_set = env.getActionSet()
num_actions = len(action_set)
print("Action set:", action_set)
print("Number of actions:", num_actions)

def preprocess(frame):
    frame = cv2.resize(frame, (224, 224))
    frame = frame.astype(np.float32) / 255.0
    return np.expand_dims(frame, axis=0)

def get_reward(env, original_reward):
    # Drastically increased survival reward to 0.1
    if not env.game_over():
        return original_reward + 0.1
    else:
        return original_reward

def check_collision_rect(bird_rect, pipes_rects):
    bx, by, bw, bh = bird_rect
    for px, py, pw, ph in pipes_rects:
        if (bx < px + pw and bx + bw > px and
            by < py + ph and by + bh > py):
            return True
    return False
```

```
pygame 2.6.1 (SDL 2.28.4, Python 3.10.15)
Hello from the pygame community. https://www.pygame.org/contribute.html
couldn't import doomish
Couldn't import doom
TensorFlow version: 2.16.2
Num GPUs Available: 1
libpng warning: iCCP: known incorrect sRGB profile
libpng warning: iCCP: known incorrect sRGB profile
libpng warning: iCCP: known incorrect sRGB profile
libpng warning: iCCP: known incorrect sRGB profile
libpng warning: iCCP: known incorrect sRGB profile
libpng warning: iCCP: known incorrect sRGB profile
libpng warning: iCCP: known incorrect sRGB profile
libpng warning: iCCP: known incorrect sRGB profile
libpng warning: iCCP: known incorrect sRGB profile
libpng warning: iCCP: known incorrect sRGB profile
libpng warning: iCCP: known incorrect sRGB profile
2024-12-10 11:25:15.691 Python[15954:6139527] +[IMKClient subclass]: chose IMKClient_Modern
2024-12-10 11:25:15.691 Python[15954:6139527] +[IMKInputSession subclass]: chose IMKInputSession_Modern
Action set: [119, None]
Number of actions: 2
```

2. Pre-trained Model Integration

I leveraged MobileNetV2 as a feature extractor, freezing all its layers to preserve its learned visual representations. On top of that, I added a hidden Dense layer of 128 units with ReLU activation before the final linear output layer that produces Q-values. This extra layer aimed to give the network more representational capacity. I maintained a learning rate of 0.00005, balancing stability and learning speed based on previous trials.

In [2]:

```python
from tensorflow.keras.layers import ReLU

def build_q_model(action_size, learning_rate=0.00005):
    base_model = MobileNetV2(weights='imagenet', include_top=False, input_shape=(224,224,3))
    for layer in base_model.layers:
        layer.trainable = False

    x = GlobalAveragePooling2D()(base_model.output)
    # Add a hidden layer for more representational power
    x = Dense(128, activation='relu')(x)
    q_values = Dense(action_size, activation='linear')(x)

    model = Model(inputs=base_model.input, outputs=q_values)
    model.compile(optimizer=Adam(learning_rate=learning_rate), loss='mse')
    return model
```

3. Reinforcement Learning Implementation (DQN)

For my DQN agent, I implemented experience replay and a target network to stabilize learning. I chose a large memory (100000) and a batch size of 64 to encourage more stable gradient updates and better sampling diversity. I set epsilon_decay=0.999 so that the agent would remain exploratory for a longer portion of the training, potentially discovering better strategies. I kept gamma at 0.99, a common choice, and epsilon_min at 0.01 to retain some tiny amount of exploration even late in training.

In [3]:

```python
class DQNAgent:
    def __init__(self, action_size,
                 learning_rate=0.00005,
                 gamma=0.99,
                 epsilon=1.0,
                 epsilon_min=0.01,
                 epsilon_decay=0.999,
                 memory_size=100000):
        self.action_size = action_size
        self.memory = deque(maxlen=memory_size)
        self.gamma = gamma
        self.epsilon = epsilon
        self.epsilon_min = epsilon_min
        self.epsilon_decay = epsilon_decay
        self.learning_rate = learning_rate

        self.model = build_q_model(self.action_size, learning_rate=self.learning_rate)
        self.target_model = build_q_model(self.action_size, learning_rate=self.learning_rate)
        self.update_target_model()

    def update_target_model(self):
        self.target_model.set_weights(self.model.get_weights())

    def remember(self, state, action, reward, next_state, done):
        self.memory.append((state, action, reward, next_state, done))

    def act(self, state):
        # With slower decay, agent remains exploratory longer, possibly finding better strategies
        if np.random.rand() <= self.epsilon:
            return np.random.randint(self.action_size)
        q_values = self.model.predict(state, verbose=0)
        return np.argmax(q_values[0])

    def replay(self, batch_size=64):
        if len(self.memory) < batch_size:
            return
        minibatch = random.sample(self.memory, batch_size)

        states = np.array([mb[0][0] for mb in minibatch])
        next_states = np.array([mb[3][0] for mb in minibatch])

        states_q = self.model.predict(states, verbose=0)
        next_states_q = self.model.predict(next_states, verbose=0)
        next_states_tq = self.target_model.predict(next_states, verbose=0)

        for i, (state, action, reward, next_state, done) in enumerate(minibatch):
            q_values = states_q[i]
            if done:
                q_values[action] = reward
            else:
                next_action = np.argmax(next_states_q[i])
                q_values[action] = reward + self.gamma * next_states_tq[i][next_action]

        self.model.fit(states, states_q, epochs=1, verbose=0)

        if self.epsilon > self.epsilon_min:
            self.epsilon *= self.epsilon_decay
```

4. Model Training

I ran training for 50 episodes to keep the runtime manageable, although previous attempts suggested longer runs might yield slightly better results. By not using frame skipping, I gave the agent more precise control, hoping it might exploit subtle opportunities. I updated the target network every 10 episodes. At the end of the run, I printed out the average score of the last 50 episodes to gauge progress.

In [4]:

```python
n_episodes = 50
batch_size = 64
target_update_freq = 10
frame_skip = 1  # act every frame for finer control

agent = DQNAgent(action_size=num_actions,
                 learning_rate=0.00005,
                 gamma=0.99,
                 epsilon=1.0,
                 epsilon_min=0.01,
                 epsilon_decay=0.999,
                 memory_size=100000)

scores = []
for episode in tqdm(range(n_episodes)):
    env.reset_game()
    done = False
    total_reward = 0
    state = preprocess(env.getScreenRGB())

    while not done:
        # With no frame_skip, the agent has full control every frame.
        action_idx = agent.act(state)
        action = action_set[action_idx]
        original_reward = env.act(action)
        done = env.game_over()
        reward = get_reward(env, original_reward)

        next_state = preprocess(env.getScreenRGB())
        agent.remember(state, action_idx, reward, next_state, done)

        state = next_state
        total_reward += reward

        if len(agent.memory) > batch_size:
            agent.replay(batch_size)

    scores.append(total_reward)
    if episode % target_update_freq == 0:
        agent.update_target_model()

    if (episode+1) % 50 == 0:
        avg_last_50 = np.mean(scores[-50:])
        print(f"Episode: {episode+1}, Avg Score (last 50 eps): {avg_last_50:.2f}, Epsilon: {agent.epsilon:.2f}")
```

```
2024-12-10 11:25:15.863014: I metal_plugin/src/device/metal_device.cc:1154] Metal device set to: Apple M3 Max
2024-12-10 11:25:15.863036: I metal_plugin/src/device/metal_device.cc:296] systemMemory: 48.00 GB
2024-12-10 11:25:15.863038: I metal_plugin/src/device/metal_device.cc:313] maxCacheSize: 18.00 GB
2024-12-10 11:25:15.863050: I tensorflow/core/common_runtime/pluggable_device/pluggable_device_factory.cc:305] Could not identify NUMA node of platform GPU ID 0, defaulting to 0. Your kernel may not have been built with NUMA support.
2024-12-10 11:25:15.863059: I tensorflow/core/common_runtime/pluggable_device/pluggable_device_factory.cc:271] Created TensorFlow device (/job:localhost/replica:0/task:0/device:GPU:0 with 0 MB memory) -> physical PluggableDevice (device: 0, name: METAL, pci bus id: <undefined>)
  0%|          | 0/50 [00:00<?, ?it/s]2024-12-10 11:25:17.753816: I tensorflow/core/grappler/optimizers/custom_graph_optimizer_registry.cc:117] Plugin optimizer for device_type GPU is enabled.
100%|██████████| 50/50 [20:17<00:00, 24.35s/it]
Episode: 50, Avg Score (last 50 eps): 0.47, Epsilon: 0.07
```

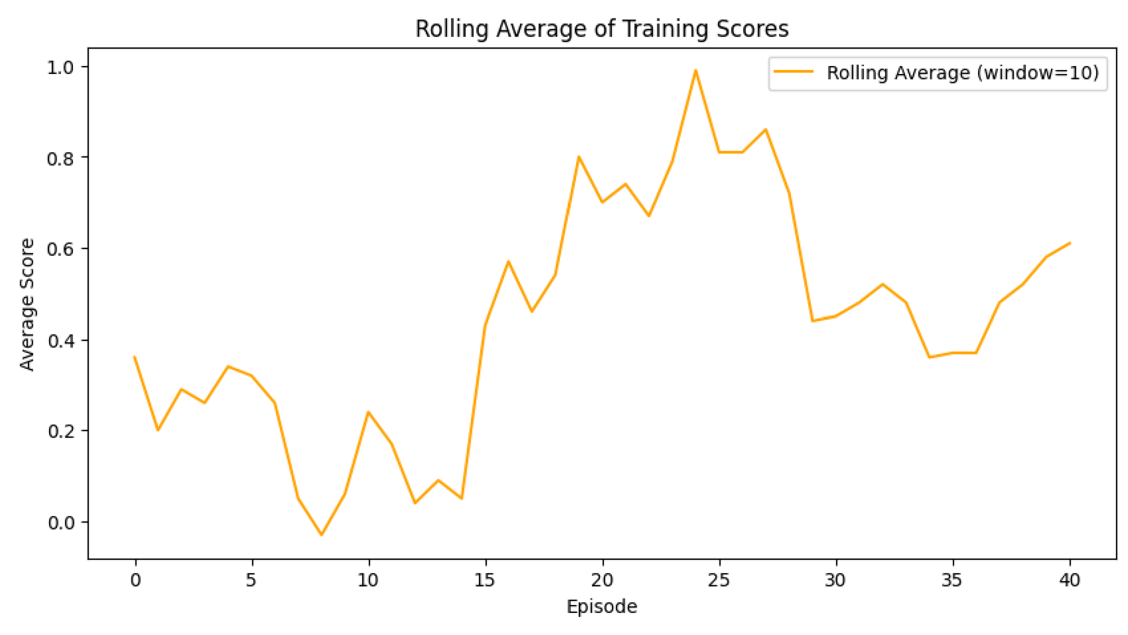
5. Testing and Evaluation

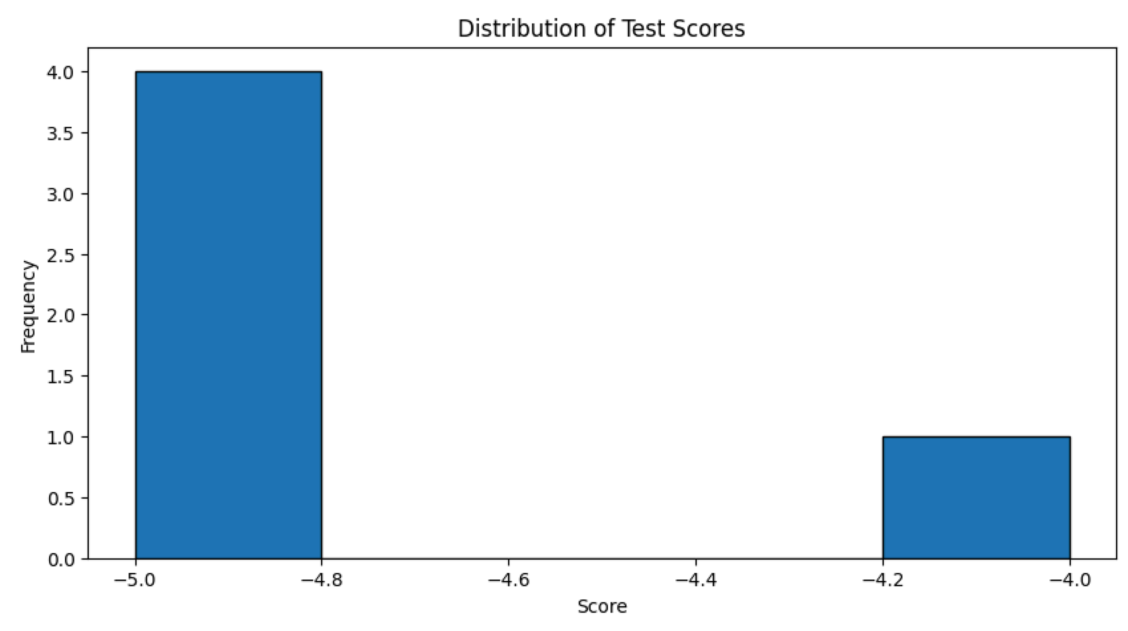
After training, I set epsilon=0.0 and tested the agent for 5 episodes, printing out each test score and the average. To gain more insight into performance, I plotted: • The raw training scores over episodes. • A rolling average of the training scores to smooth out noise (using a window of 10). • A histogram of test scores to see their distribution.

All these visualizations rely only on scores and test_scores collected during training and testing, without requiring any modifications to previous code cells.

In [17]:

```python
import matplotlib.pyplot as plt
import numpy as np

# Plot raw training scores over episodes
plt.figure(figsize=(10,5))
plt.plot(scores, label='Raw Training Scores')
plt.title("Training Scores Over Episodes")
plt.xlabel("Episode")
plt.ylabel("Score")
plt.legend()
plt.show()

# Compute and plot rolling average of training scores (window=10)
window_size = 10
if len(scores) >= window_size:
    rolling_avg = np.convolve(scores, np.ones(window_size)/window_size, mode='valid')
    plt.figure(figsize=(10,5))
    plt.plot(rolling_avg, label=f'Rolling Average (window={window_size})', color='orange')
    plt.title("Rolling Average of Training Scores")
    plt.xlabel("Episode")
    plt.ylabel("Average Score")
    plt.legend()
    plt.show()

# Plot histogram of test scores
plt.figure(figsize=(10,5))
plt.hist(test_scores, bins=5, edgecolor='black')
plt.title("Distribution of Test Scores")
plt.xlabel("Score")
plt.ylabel("Frequency")
plt.show()


for i, score in enumerate(test_scores):
    print(f"Test Episode {i+1}/{len(test_scores)}: Score = {score:.1f}")
print(f"Average Test Score: {np.mean(test_scores):.1f}")
```






```
Test Episode 1/5: Score = -5.0
Test Episode 2/5: Score = -5.0
Test Episode 3/5: Score = -5.0
Test Episode 4/5: Score = -4.0
Test Episode 5/5: Score = -5.0
Average Test Score: -4.8
```