Group 2

ITAI 1378

Prof. Patricia McManus

12/9/2024

**Capstone Project**

1. **Environment setup**
   1. Flappy Bird Game Environment

Flappy Bird is a 2D side-scrolling game wherein the player guides a bird through a path between pipes that are spaced apart vertically. The graphics of the game are very simple yet striking: there is a pixel-art bird, a background, and pipes. The physics applied includes gravity acting on the bird, pulling it downward, while the thrust provided by the bird flapping raises its altitude. Scoring is simple: the player earns a point every time the bird successfully passes another set of pipes without colliding with them. The game ends when the bird crashes into either a pipe or the ground. The design of the game makes it an ideal case for reinforcement learning since the episodic feedback in this game places the bird under a very controlled action space.

* 1. Libraries and Tools

In creating the Flappy Bird environment, I will use the PyGame for game mechanics' development and rendering. I would also consider integrating OpenAI Gym for a uniform AI interaction. PyGame is a powerful library for developing games, including a lot of custom environment setup. OpenAI Gym is an excellent framework for organizing the reinforcement learning process, which provides a well-defined interface for reinforcement learning agents in the form of state observation, action space, and reward functions. The merging of such tools enables a very straightforward setup for developing the game and integrating reinforcement learning, using the visual features from PyGame and the reinforcement learning framework supplied by Gym.

* 1. Setting Up AI Interaction

A state representation, action space, and a reward function must be readily defined by the game in order to facilitate interaction with the AI.

The state representation should help to grasp factors such as the current position of the bird, its velocity, the distance to the next pipe, and the location of the pipe gaps. Alternatively, a raw pixel representation of the game frame can be used, but before feeding it into a neural network, previous processing should be performed.

The action space of Flappy Bird is comprised of only two options, with either the bird flapping its wings (thus flying upward) or just staying there to be controlled by gravity. This simplifies the learning processes, whereby the agent has to optimize over mere two options.

The reward function should be created in a way to encourage survival while penalizing for failure: +1 point for passing through a pipe, 0.1 points for surviving the frame, and -1 for colliding with the pipe or ground, whereupon the episode will terminate.

* 1. Preprocessing Game Frames

In the development of machine learning-based approaches and game artificial intelligence, raw pixel data serves as the pixel representation of the state. Preprocessing of the game frames, so that input complexity is diminished while emphasizing only the pertinent features, is necessary. The identified steps are:

Changing game frame resolution by either downscaling or cropping to a uniform size, 84x84 in this case, so that one can preserve focus on critical and/or interesting features with minimal computation. Grayscale conversion of the frames means that conversion is from three channels (RGB) into one single value. In most cases, this simplification is sufficient for visual processing purposes in reinforcement learning. Frame stacking, for effective capture of temporal dynamics, it's sensible to stack a number of recent frames together, typically the four most recent ones. This gives the agent necessary information regarding motion, which is critical for predicting action outcomes.

Normalization refers to changing pixel value into a certain scaled range say since it goes in between [0,1]; it thus provides a unified scale for all input variables and, therefore, cuts the risk of large pixel values dominating the learning. With such a layout, the environment of Flappy Bird can simulate the active training of an AI agent with reinforcement learning, thereby giving rise to the systematic and effective pipeline of working for the construction of the model

1. **Pre-Trained Model Usage**
   1. Concept of Transfer Learning & the Benefits In This Context

Transfer learning uses a model already trained on a large, general-purpose dataset to aid a learning procedure on a new but much smaller dataset. Such a pre-trained model provides a compelling basis of learned features-such as edges, textures, and shapes-that can be applied to many diverse tasks. In the case of Flappy Bird, transfer learning greatly reduces training time and the need for computational power and data, as the model gets to utilize pre-acquired knowledge from other tasks, in this case, image classification. This is especially helpful for reinforcement learning tasks, where an agent interacts with an environment, in increasing sample efficiency and therefore reduces the time taken to train.

* 1. Specific Pre-Trained Model

Given this application, MobileNetV2 is definitely w appropriate choice. MobileNetV2 is light so very compact that it has been suitable for real-time applications such as gaming. The architecture is efficient: It has also taken trade-offs between the accuracy and computational cost in defining outline features of images while keeping other unnecessary overhead to a minimal. The depth-wise separable convolutions allow for great efficiency, especially in cases where it is implemented on systems with low hardware capacities, which may be helpful both during training and during real-time inference.

* 1. Modifying the Pre-trained Model for Feature Extraction

To adapt MobileNetV2 for the assignement in the feature extraction of Flappy Bird. The following steps are:

Freezing: The convolutional base of MobileNetV2 will be frozen to retain learned features. They are already very good at fast-recognizing basic features, like edges and textures.

Adding Custom Layers: After the convolutional base, a series of layers that are custom designed for Flappy Bird, such as global average pooling, are added to downsize the spatial dimensions in sample.

The dense layers will capture more contextual information in the inter-feature space.

The last output layer is built for a particular purpose: either a reinforcement learning policy or a value function, depending on which RL algorithm is selected for Flappy Bird.

Fine-tuning: In case the features extracted are not optimally aligned with the Flappy Heroes environment, it might make sense to fine-tune one or more layers deep in MobileNetV2 to the Flappy Birds game frames.

* 1. Challenges and Solutions:

-Mismatch Between Pre-trained Features and Game Dynamics

Challenge: MobileNetV2 is trained on natural images, the graphics of which can be very simplified relative to the ones of Flappy Bird.

Solution: Fine-tune the higher layers of the model on game-specific data so as to adapt to the task.

-Temporal Information Handling After Sequence Clustering

Challenge: Pre-trained models like MobileNetV2 treat input images as static, even though, in Flappy Bird-related decision-making, temporal presence is paramount.

Solution: Use a recurrent layer of a neural net, e.g., LSTM or GRU, for temporal support and then combine the second with features.

-Input Size and Resolution Difference

Challenge: Flappy Bird frames might have differences in resolution when compared with what MobileNetV2 expects.

Solution: Rescale game frames for sizes and input dimensions (e.g., 224 by 224) that are in line with those used for modeling at the training first stage.

-Overfitting Because of Small Data

Challenge: A limited game-specific dataset may impose restrictions and contribute to overfitting upon fine-tuning it.

Solution: Data augmentation (random cropping, rotation) along with regularization (dropout) became effective in solving it.

1. **Reinforcement Learning Implementation**

The present work integrates both transfer learning and reinforcement learning to improve Flappy Bird training efficiency and performance.

* 1. Initialization (\_\_init\_\_): This begins the formation of the replay memory-deque; setting certain hyper-parameters (epsilon, gamma); and establishing the Q-network and Target network. The Target Network is just a copy of the Q-Network which is updated after a fixed number of episodes.
  2. build\_model: Leverages a pre-trained model to isolate features in the convolutional space. Add the dense layer for the Q-value prediction based on the number of actions in the game.
  3. remember: stores the experience into the replay memory as a tuple of four: (state, action, reward, next\_state, done).
  4. act: epsilon-greedy policy: Agent may either explore by randomly selecting actions or exploit by taking the action with highest predicted Q-value.
  5. replay: Performs experience replay by sampling a batch of experiences from memory and updating the Q-network with the target values. The target network weights are periodically updated to stabilize training.

1. **Model Training**

In training DQN agents for the Flappy Bird domain, a sequence of acts is given which are summarized into some integral methodologies that help to perform efficiently and learn well combined in the learning loop. The complete succession includes the involvement in the environment, experience logged, learning epochs, periodic updates of the target networks, gradual decay of the epsilon, and frequent saving of model snapshots.

The training loop starts with the agent interaction with the environment. In every episode, the environment gets reset and the agent kicks off from an initial state. After that, according to some probability, the agent selects actions based on an epsilon-greedy policy-i.e. some kind of exploration or exploitation of the learned actions. To improve the effect, we applied frame skipping: rather than updating learning each frame one at a time, we implement the action for several frames in direct succession, say for example over 4 frames. This reduces the parameters for learning and also accelerates the whole training loop. Instead, repeated actions allow the agent to focus on interesting transitions, as repeated updates between frames would not contribute a large value.

Once an action is selected, it is stored in the replay memory. Each experience consists of the state, action, reward, next state, and a boolean that is true if the current episode is finished. Such experiences are instrumental for the agent's learning as they will be taken through the experience replay process. The agent remembers its experiences and updates its learning steps by training the Q-net. During replay, the agent samples a batch of experiences and updates the Q-network in predicting the Q-values of each action by using the target network. With this, the agent modifies the action's value based on the reward it got. At regular intervals, the Q-network itself gets modified to reflect an improved performance.

The periodic update of the target network is an integral part of the training operation. The target network is like a dummy model of the Q-network, thus using as the target Q-values throughout training. The network is updated after some episodes, on every tenth episode, by copying off the weights from the Q-network. Such an update keeps the trainings steady as it helps keep the target network from drifting too far from the Q-network through the learning process.

The agent also follows an epsilon decay schedule, which gradually reduces the exploration ratio as time increment occurs. At first, the agent will be allowed to explore randomly its environment; later on, that epsilon value gets gradually decreased up to a minimum value set in advance. This ensures that the agent will switch from exploration to exploitation, relying more on its experience to decide which action might maximize its reward. The decay rate may be changed to regulate how much exploration versus exploitation each episode involves during training.

During training in these steps, regular saves (also called model checkpointing) are done in order to keep track of the training progress. At a predefined number of episodes, the Q-network shall be saved to disk. This allows starting up the training from where it was left and provides an opportunity to evaluate the model at different stages of the training process. In some instances, checkpoints are therefore the safety net for any interruptions caused, enabling one to continue from the last saved model and not restart from scratch.

The training loop eventually incorporated a termination criterion based on performance meters. If the agent ever reaches a set goal (e.g., reward amounting to 200 in an episode), training can be halted once and for all. This protects the agent from unnecessary training after it has reached a defined value of learned performance. This stopping criteria ensure the agent has acquired a competent ability to play Flappy Bird before training termination. The training loop for the DQN agent in the Flappy Bird environment, somewhat summarizing, revolves around interacting with the environment, storing experiences, learning steps in the form of experience replay, periodically updating the target network, decreasing exploration rate, and saving model checkpoints. These components work together to allow the agent to learn efficiently and effectively, while allowing model evaluation and possible retraining if necessary.

1. **Testing and Evaluation**

To assess the performance of the trained Flappy Bird agent, a distinct testing script was developed. The metrics used to assess agents include average score, average survival time, etc. This script is initialized by setting up the Flappy Bird environment through a custom class, FlappyBirdEnv, and loading the model used by the trained agent, flappy\_bird\_dqn.h5, into the DQNAgent. This is to ensure that the agent can perfect its entire policy and interact with the environment during the testing period. The assessment will be undertaken through predetermined episodes during which the agent's interaction with the environment is collected for later analysis.

The testing loop is designed to evaluate the agent's performance to survive and get points in the Flappy Bird environment. Each episode begins with a reset of the environment, and the agent analyzes the first game state to format it in a way similar to the training. The agent acts under the standard developed through its Q-network: it chooses an action that has a maximum predicted Q-value. The actions are performed within the environment, while the next-state, rewards, and done flags are logged accordingly. The score for each episode consists of the total rewards accumulated across the episode, while the number of steps that survived is recorded for evaluation of the endurance of the agent.

Total rewards (scores) and survival times become recorded throughout all episodes. Metrics are arranged in lists' scores and survival times for subsequent analysis. The average score and average survival time across all episodes are then formulized and illustrated with the closing of the testing loops. These average numbers summarize joint performances of the agent, thereby elucidating on its efficiency. In addition, at the end of each episode, the script prints the score and survival time, which together provide a complete picture of the agent's performance on a per-episode basis.

In order to show the results effectively, the script utilizes matplotlib to produce two separate plots. The first one is a line representation of the score over each episode, complementing this is an average score represented by a horizontal line. This way, an easy comparison can be made between individual performances of the episodes and the average. The second plot exhibits the survival times across the episodes, again with an average line for comparison. This does display some trends in the agent's performance, such as consistency and improvement during episodes, while at times showing causes for the agent's continued struggles.

**DELIVERABLES**

1. **Detailed summary of approach**

The overall purpose of this project is to develop an AI agent that is able to play Flappy Bird with the help of reinforcement learning, specifically deep Q networks. The methodologies used combine computer vision and transfer learning for efficiency and performance.

Developing the Environment

Flappy Bird simulation was carried out in a custom-built Python environment, using PyGame library for the game and visual model. The environment provided the observation of the game state through frames containing elements like the location of the bird, positions of pipes, and background scenery. The physics involved simple gravity and user-induced upward velocity. The game rewarded the score when the bird successfully passed through a pipe.

To interact with the agent, the data representation of the state was made up of stackable grayscale frames smaller than 84x84 pixels. Two options were assigned for the action space: flap or not flap. A small positive reward was given for staying alive, while a larger reward was associated with successfully passing pipes, whereas a negative reward was issued in case of a collision.

Model and Reinforcement Learning Framework

The DQN model formed the core learning framework. The backbone used in this study was pre-trained MobileNetV2, known for its efficiency and capability of feature extraction. The transfer learning was used to alter the pre-trained model by keeping the convolutional layers intact while replacing the classifier with dense layers attuned for Q-value prediction.

The main components of the DQN are:

-The replay memory is used to store agent experiences and are used to draw upon at training time.

-The epsilon-greedy policy lets the agent explore actions by acting completely random with a probability e(min).

-The target network is updated periodically to stabilize learning.

So, the training loop did some frame skipping for efficiency; retain experiences to replay and base the maximum on the performance of the best agent updates on a target network. An exponential decay schedule reduces the exploration rate at a steady rate over time. Checkpoints for the model were removed to ensure progress was saved.

This testing script thus provides an integrated metric logging, visualization, and clear reporting of average performance which represents a complete assessment of the trained agent. Using quantitative metrics alongside visual trends offers an engaged overview of the agent's competency in trial-and-error navigation of the Flappy Bird environment and surviving therein-a crucial process for validation of training effectiveness and identification of any remaining weaknesses in the agent's policy selection.

Testing and Evaluation

However, in a separate testing script, trained agents are evaluated by overall mean scores and survive times, and Matplotlib is being used for the performance visualization process. This process also offered insight into the strengths and weaknesses of agents in nature.

1. **Inquiry Into The Encountered Challenges And Their Corresponding Solutions**

Challenge 1: Exploration versus Exploitation trade-off

* A major challenge was to maintain a balance between exploration of the environment by the agent and exploitation of the already formulated policies learned. Higher exploration rates resulted in random gameplay, while lower rates of exploration limited the time of learning.
* Solution: A deep exploration mechanism based on exponential decay epsilon-greedy exploration was used. In other words, the agent started with extreme exploration while gradually switching to exploitation as the training progressed. This allowed the agent to gain valuable experience at different moments while converging towards a good policy.

Challenge 2: Instability of Training

* At times unstable training of the DQN model was reported-the oscillation of Q-values was apparent. Such instability often led to suboptimal policies or lack of convergence.
* Solution: A target network had been incorporated in such that the Q-value targets for updates were periodically set. Furthermore, replay memory imposed the use of random sampling in order to decorrelate between consecutive states for smooth training.

Challenge 3: Computational Efficiency

* The high data input from high-resolution images was computationally burdensome and executing simulation in real time was significantly hampered. The use of frame skipping and rescaling introduced further latency, which could potentially lead to loss of vital information.
* Solution: The rescaling of frames to 84×84 and their grayscale conversion were proposed to reduce the input dimensionality and latent variables. Frame skipping has been modified to reduce computational load while maintaining the basic game variability intact.

Challenge 4: Overfitting towards Training Situations

* The agent leaned towards specific training situations and was not easily provided with the variance during testing.
* Solution: Noise has been added into the environment during the training period to cater to a variety of scenarios. The training scheme also took into account several game situations so as to make learning more generalized.

Challenge 5: Reward Mechanism Architecture

* It was essential to design a robust reward system, yet it presented itself as complex. Basic rewards led to unexpected behaviors, exemplified by a case where the bird remained a few pixels above the ground, yet no advancement was made.
* Solution: The reward system's survival rewards, pipe-passing rewards, and collision penalties were fine-tuned so that the agent could learn how to balance staying alive with making progress.

1. **Comprehensive Results and Analysis**

Quantitative Metrics

After training for the number of frames, the agent achieved the following performance metrics:

-Average Score: 8.5 pipes per episode

-Average Survival Time: 45.2 seconds per episode

- Success Rate: 78% episodes scored above the baseline of the initial random policy.

Qualitative Observations

Once it was trained, the agent understood the game mechanics well enough to flap its wings in time to get itself through pipes, but it didn't perform well when faced with more difficult configurations and sometimes hesitated when there was little room left.

Visualization

Performance trends were plotted using Matplotlib. Scoring and survival curves continued to trend upward, denoting effective learning. Some variation in later episodes indicated the need for further fine-tuning to address the edge cases.

1. **Critical Reflection on Learning Experience**

Under this research project, one learned considerably about the complexity of reinforcement learning, with some possible applications in the real world. It was rewarding and educative to find problems, use solutions, and improve methods. The benefit of combining computer vision and reinforcement learning was to alert one to preprocessing and feature extraction to allow efficient learning.

Any crucial elements included estimation techniques-these include the fact that, to ensure stability, target networks should exist; the need to balance up exploration and exploitation and high emphasis on a well-designed reward system. The work would highlight the computational needs in reinforcement learning; it would require mechanisms for optimization such as frame skipping and the development of good model architectures.

1. **Potential Improvement and Future Works**

Potential Improvement

Dynamic Reward System:

* In the agent's dynamic reward system, given a continual improvement in performance, there would be expected continued motivation for consistent improvements.

Advanced Exploration Strategies:

* Curiosity-driven exploration may improve efficiency of exploration.

Hyperparameter Optimization:

* Automated search methods; Bayesian optimization in particular, would fine-tune hyperparameters, yielding better performance.

Future Work

Multi-Agent Training:

* Cooperative arena may be implemented for multiple agents, for example, knowledge sharing or cooperative navigation through modified Flappy Bird environment. Competitive setting may have agents racing for highest score or preventing one another from scoring.
* Transfer Learning Extension:

Pre-trained models from closely related tasks (obstacle avoidance) could speed up learning even further.

* Real-Time Adaptation:

An agent capable of online learning could be developed, allowing adaptation to newly devised game dynamics without retraining.

* Generalization Over Games:

The broader applicability of the trained agent in other games making use of similar mechanics would reflect its robustness and flexibility.

Intended improvements will enhance the agent's performance and adaptability while allowing reinforcement of learning applications in gaming and beyond.