Q-learning vs. DQN: Two Approaches in Flappy Bird Reinforcement Learning

Sanindie Silva, Vanshita Uthra, Olivia Xu
Queen's University, CISC 474

Winter 2024

Introduction and Problem Formulation

Flappy Bird is a game that emerged in early 2013 and gained popularity within a year. The motive of this game is to help the bird manipulate through the pipes to stay alive. This report highlights how different algorithms affect the training in terms of the rewards generated and the number of time steps required to solve the task per episode, along with the optimal performance of the agent.

State/Observation Space: The state space in Flappy Bird is the variables that showcase the current situation of the bird. The state space of this project includes the last pipe's horizontal position, the last top pipe's vertical position, the last bottom pipe's vertical position, the next pipe's horizontal position, the next top pipe's vertical position, the next bottom pipe's vertical position, the next next pipe's horizontal position, the next next top pipe's vertical position and the next next bottom pipe's vertical position, the player's vertical position, the player's vertical velocity and the player's rotation. The size of the space can vary as it depends on the value that each state can take. Therefore, the state is continuous as the position and distance can take on a continuous range of values.

Action Space: The actions in this problem include two actions: "do nothing" and "flap". The action space is discrete as the action that can be taken is distinctive.

Reward Scheme: These are the rewards given to the agent when they perform a certain action:

$$r = \begin{cases} +0.1, & for\ every\ frame\ the\ agent\ remains\ alive \\ +1.0, & successfully\ passing\ a\ pipe \\ -1.0, & dying \\ -0.5, & touch\ the\ top\ of\ the\ screen \end{cases}$$

Algorithm Description

Q-Learning

The first algorithm that we chose to experiment with was Q-Learning. This is a model-free reinforcement learning algorithm that enables an agent to learn the optimal action-selection policy for any given finite Markov decision process by learning and updating estimates of the action values (Q-values). It operates through repeated interactions with the environment, updating the Q-values based on the rewards received for actions taken in various states. The updates are guided by the Bellman equation, combining immediate rewards with the discounted maximum future rewards expected according to the current policy, adjusted by a learning rate.

Implementation Details:

Initializing Q-Values*:* Q-values are stored in a dictionary where the keys are the states, and the values are arrays containing Q-values for each possible action in that state. When a new state is encountered, we initialize that new state entry with zeros for each possible action.

Action Selection: The algorithm uses an epsilon-greedy strategy to balance between exploration of new actions and exploitation of known actions. With probability ε, a random action is selected (exploration). With probability (1 - ε), the agent chooses the action with the highest Q-value for the current state (exploitation). We progressively decrease the value of ε to reduce exploration as the agent converges toward optimal behaviour.

Updating Q-Values*:* We calculate the temporal difference (TD), which is the difference between the estimated Q-value and the observed reward plus discounted future Q-value. This difference is then used to update the Q-value of the taken action in the current state.

Hyperparameter Selection:

- **Epsilon values:**
  - **Initial value:** Set to 1 to ensure that the agent starts by exploring actions randomly to maximize exploration

    - **Final value:** Set to 0.01. This aims to give priority to exploitation later in the training process.
    - **Decay formula:** The formula used for epsilon decay is start_epsilon / (n_episodes / 2), meaning epsilon will reach its minimum value halfway through the training process. This assumes that the first half of the episodes will provide sufficient exploration while the second half will prioritize exploitation.
- **Learning Rate:** Set to 0.5, a moderately low value. In a game like Flappy Bird, where the mechanics are relatively static (i.e., the physics and obstacle patterns don't change over time), we don't need a very high learning rate to enable the agent to adapt to changes or new patterns in the environment.
- **Discount Factor:** Set to 0.95. A discount factor of 0.95 indicates a strong preference for future rewards.

DQN

The DQN algorithm extends the classic Q-learning by using a deep neural network to approximate the Q-value function. The network predicts Q-values (expected future rewards) for each action given an input state. The primary advantages of using DQN include its ability to handle high-dimensional state spaces, which helps because the environment and state space are very dynamic and would benefit from approximation.

Implementation Details:

***Network Architecture*:** The DQN model consists of an input layer, which accepts an input with a shape equal to state_dim (the dimensions of the environment's state space), followed by two hidden layers, with 128 and 256 neurons respectively, both with a ReLU activation. Finally, the output layer contains action_dim neurons, each representing the Q-value of a possible action in the given state.

State Processing: The state from the environment (as described in **State/Observation Space**) is used directly as input to the network. No discretization was needed.

Action Selection: Implements an ε-greedy policy similar to that in Q-Learning.

Experience Replay: The agent stores experiences (state, action, reward, next_state, done) in a memory buffer to break the correlation between consecutive learning updates. Random batches from this buffer are used to train the network.

Learning Updates: Utilizes the Bellman equation to update the Q-values. The loss is calculated as the mean squared error between the currently estimated Q-values and the target Q-values derived from the Bellman equation.

Hyperparameter Selection:

- **Learning Rate (learning_rate):** Set to 0.01 initially; affects how much the neural network weights are updated during training.
- **Discount Factor (discount_factor):** Set at 0.95, which determines the importance of future rewards.
- **Epsilon Values:**
    - **Initial (epsilon):** Starting at 0.3 to allow for significant exploration at the beginning.
    - **Minimum (epsilon_min):** Decreases to 0.01, ensuring that even late in training, there is some chance of random action selection.
    - **Decay Rate (epsilon_decay):** The rate at which ε is decreased, encouraging more exploitation of learned values as training progresses.
- **Memory Buffer (buffer_size):** Set to 10000 to store a substantial number of past experiences.
- **Batch Size (batch_size):** The size of the batch used for training is 32, balancing the speed of learning updates and the generalization ability of the network.

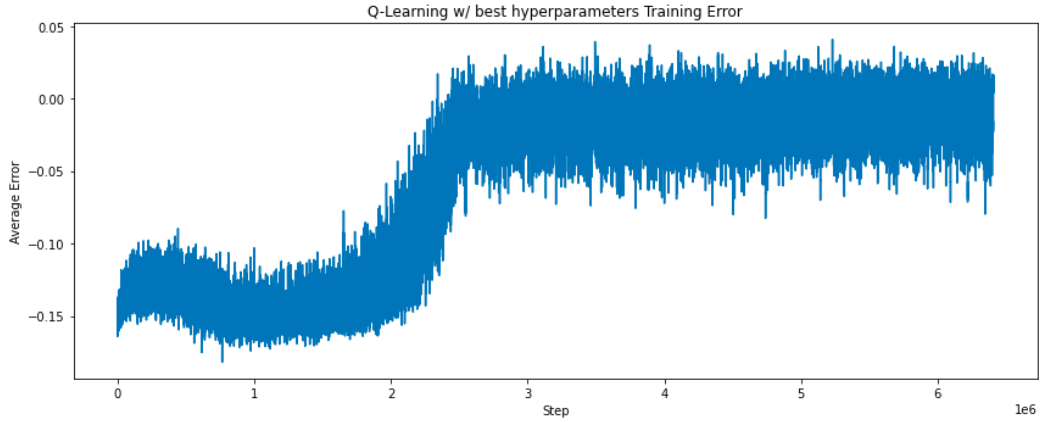
Results

Q-Learning

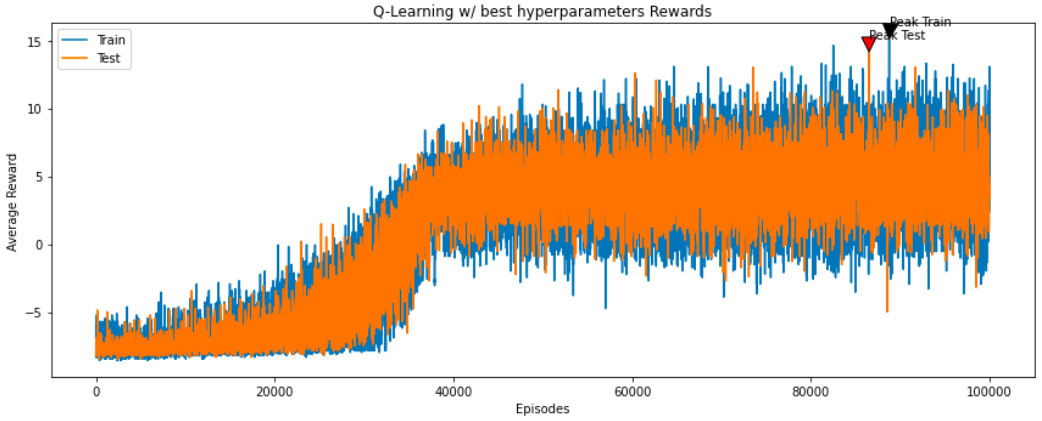
Hyperparameter tuning: We implemented grid search to find the optimal combination of hyperparameters after 5000 episodes of training with each combination. The best hyperparameters found are {'learning_rate': 0.5, 'discount_factor': 0.95, 'final_epsilon': 0.01}.

Overview of performance: After training with the best hyperparameter combination for 10,000 episodes, we obtain the following results:

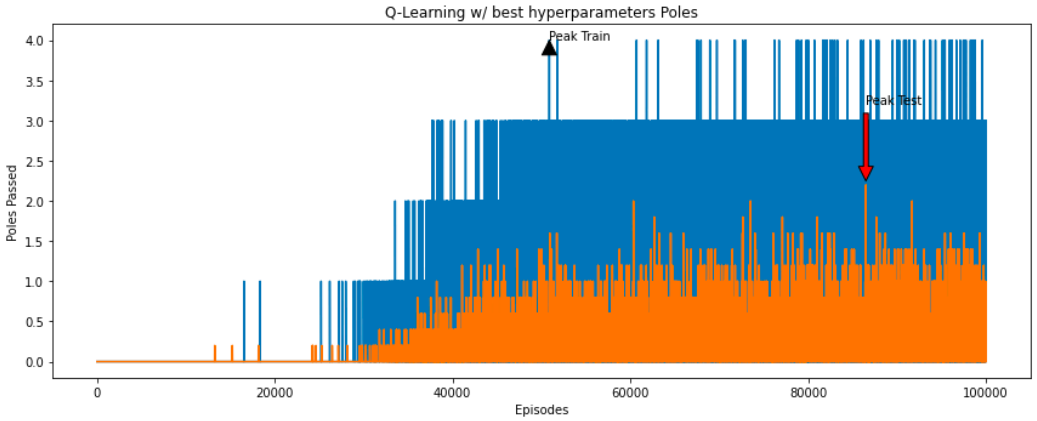
Training Error: The training error is obtained by calculating the temporal difference in each step. The graph below shows an improvement over time since the training error starts to oscillate around zero after around 2.5*10^6 steps. However, the graph also shows a fair amount of variance in training error that could indicate the algorithm's exploration of new strategies or encountering states that it has not learned to handle well.



Rewards: After training for 10 episodes, we run the policy for 5 episodes and record the average rewards. The graph below shows that testing rewards have a smaller range of values than training rewards, indicating that during testing, the policy is exploiting the knowledge gained during training without further exploration. This might lead to consistently lower rewards if the policy has not fully learned the optimal actions for all states.



Number of poles passed: We also looked at the number of poles passed as the training proceeded. The varied heights of the bars suggest that the performance was inconsistent. This inconsistency could be due to the exploration behaviour inherent in Q-learning, where the agent sometimes makes suboptimal choices to learn more about the environment. However, we do see a gradual increase in the number of poles passed, meaning that the policy obtained from Q-Learning has been improving.

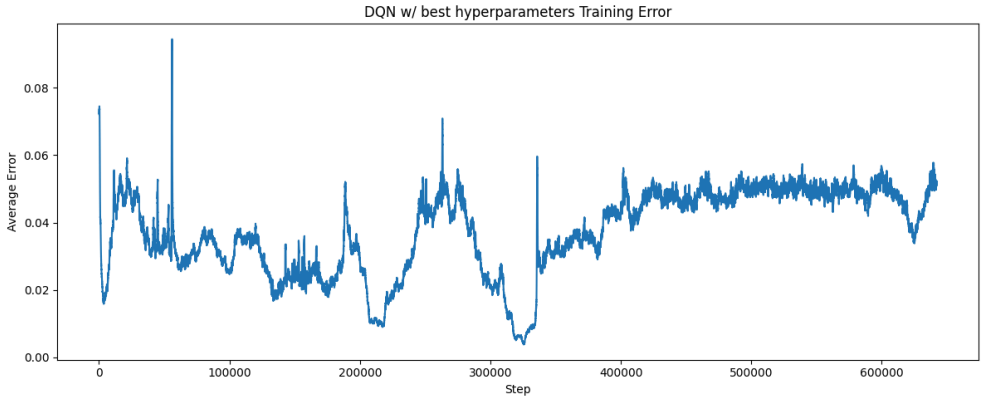


DQN

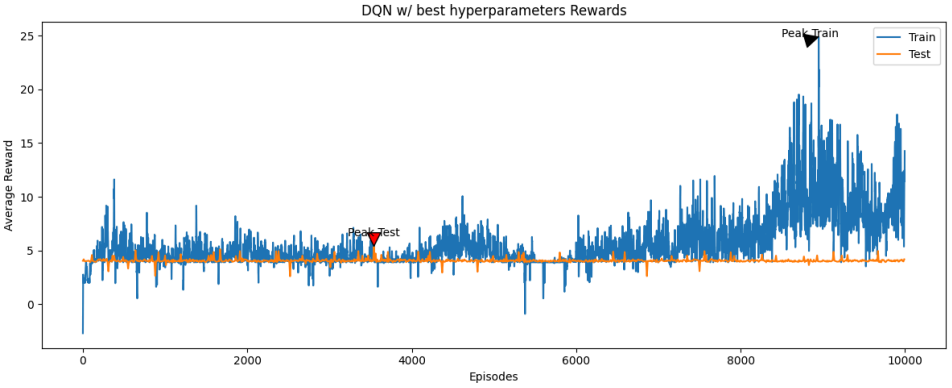
Hyperparameter tuning: Using the Optuna library (which helps perform a random search for hyperparameter optimization), the best hyperparameters are: {'learning_rate': 0.002, 'discount_factor': 0.93, 'epsilon_decay': 0.94, 'batch_size': 128}.

Overview of performance: Similar to Q-Learning, we obtained the graphs showing training error, average rewards and number of poles passed:

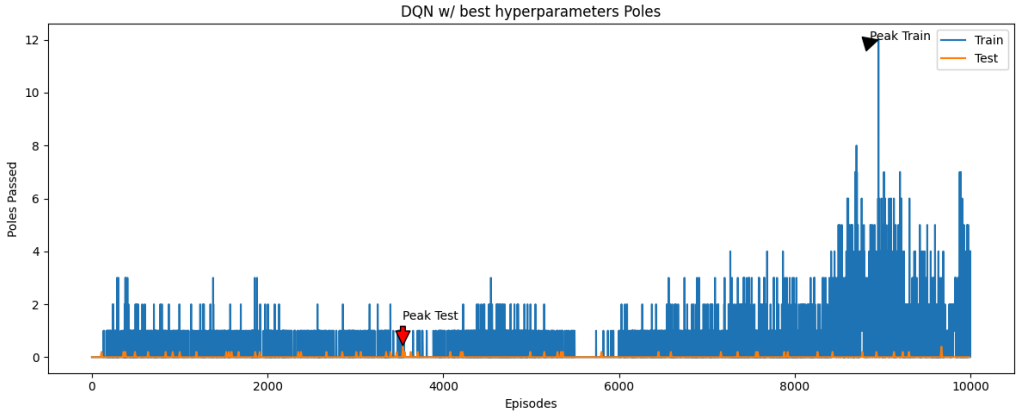
Training error: The graph below shows fluctuations and spikes, suggesting variability in the learning process, which is normal as the DQN encounters new situations and refines its predictions. The overall trend does not show a clear decrease, which could indicate that the network has not yet fully converged or that the learning rate might need adjustment.



Rewards: In the graph below, the training reward (blue line) shows considerable volatility but an upward trend, which indicates learning progress. However, the trend does not seem to plateau, which might imply that the DQN has not fully stabilized its learning. The testing rewards (orange) are more stable but consistently lower than the training rewards, which may suggest that the policy is not generalizing as well as one might hope from training to testing environments.



Number of poles passed: A similar pattern to the rewards graph is observable here, with the blue bars (training) showing that the agent has episodes of high success mixed with lower success. The testing results (orange) display lower peaks, which reinforces the idea that the policy may not be generalizing well from training to testing.



Performance Comparison

It is evident that the DQN model outperforms the Q-learning model, as expected, in navigating the complex dynamics of the Flappy Bird game environment. The DQN model leverages deep neural networks and experience replay to overcome the limitations of traditional Q-learning, resulting in more robust and efficient learning.

Comparing the Q-learning model with the DQN model in the context of the Flappy Bird task sheds light on their respective strengths and weaknesses. Q-learning, known for its simplicity, proves to be an accessible choice for basic reinforcement learning tasks due to its straightforward implementation and comprehension. Additionally, its tabular representation incurs low computational costs, making it efficient for scenarios where computational resources are limited or computational efficiency is paramount. However, Q-learning's limitations become apparent when applied to complex tasks like Flappy Bird. It struggles to generalize across the high-dimensional state space inherent in Flappy Bird, hampering its ability to learn complex strategies efficiently. Moreover, Q-learning's reliance solely on epsilon-greedy exploration may lead to suboptimal policies or inefficient exploration strategies.

On the other hand, the DQN model offers notable advantages over Q-learning in handling the complexities of Flappy Bird. By employing deep neural networks for function approximation, DQN achieves superior generalization across the high-dimensional state space inherent in Flappy Bird. Furthermore, DQN dynamically balances exploration and exploitation through epsilon decay, facilitating effective learning in challenging environments. However, the DQN model is not without its drawbacks. Its reliance on deep neural networks introduces complexity and computational overhead, making it less suitable for straightforward tasks. Additionally, the sensitivity of DQN's performance to hyperparameters necessitates meticulous tuning and experimentation to achieve optimal results.

In conclusion, while both methods have their strengths and weaknesses, the DQN model demonstrates superior performance in tackling the challenges posed by the Flappy Bird environment. Its ability to learn from raw pixel inputs, generalize across diverse states, and balance exploration and exploitation effectively makes it a powerful tool for complex reinforcement learning tasks. However, it's essential to acknowledge the computational cost

and hyperparameter sensitivity associated with DQN. For complex tasks like Flappy Bird with high-dimensional state spaces, DQN offers superior performance by leveraging deep neural networks for function approximation.

Discussion

Challenges

One of the significant challenges faced in the Flappy Bird game is a massive state space due to greater dimensionality, randomness and sparse rewards. Firstly, there are various components in the game's environment, such as the last pipe's horizontal/vertical position, the next pipe's horizontal/vertical position and the player's position and velocity. As these factors are present in every episode, this leads to having a greater number of possible states. Secondly, the pipe heights and frames are randomized after every episode, forcing the agent to adapt to the new environment constantly. Lastly, the agent receives rewards when it successfully passes through the pipe. As the agent is not always able to pass through the pipes, has to adapt to new environments constantly and has various components to the state space, it causes slower convergence and longer training times.

Future Work or Improvement

If further optimization or adaptation to specific traits of the Flappy Bird environment were needed, methods such as Double DQN, Dueling DQN or prioritized experience replay could be explored to potentially improve learning stability and performance.

Traditional DQN tends to overestimate action-values significantly which results in unstable training and low quality policy. The Bellman equation updates the Q-function, which finds the estimated value for each action's maximum expected future reward. Thus, Double DQN addresses the problem by separating action selection and action evaluation in the Q-learning update. In this method, the action which exhibits the maximum Q-value is chosen using one network ("selection network") and a separate network is used to evaluate the Q-value for that action ("target network"). Thus, Double DQN is an effective method as it reduces the overestimation of Q-values and improves learning stability, especially for games like Flappy Bird.

The Dueling DQN method separates the represented state-valued and state-dependent action through two separate streams. Through this separation, the duelling architecture can learn states which are or are not valuable, avoiding learning the outcome of each action for each state. Additionally, this method reduces the variance in Q-value methods and improves learning stability. In terms of Flappy Bird, this method can provide more accuracy when evaluating state-action pairs, leading to a more efficient performance.

Prioritized Experience Replay allows replaying transitions with high expected learning progress by measuring the magnitude of the temporal-difference (TD) error. Evaluating experiences with high TD errors allows the agent to focus on learning transitions which are more informative. Even though this leads to a loss in diversity due to stochastic prioritization and introduced bias, this can be solved through importance sampling. Stochastic sampling interpolates between pure greedy prioritization and uniform random sampling. The bias introduced in this method can be corrected using importance-sampling (IS) weights, which are then used in the Q-learning update. This method improves the learning efficiency of the agent by prioritizing experiences that are more relevant for policy improvement which leads to better performance, even for environments like Flappy Bird.

References

Dilith Jayakody. (2022, December 24). *Double deep Q-networks (DDQN) - a quick intro (with code)*. . https://dilithjay.com/blog/ddqn

Yoon, C. (2019, July 17). *Double deep Q networks*. Medium. https://towardsdatascience.com/double-deep-q-networks-905dd8325412

Yoon, C. (2019b, October 20). *Dueling Deep Q Networks*. Medium. https://towardsdatascience.com/dueling-deep-q-networks-81ffab672751

Papers with code - prioritized experience replay explained. Explained | Papers With Code. (n.d.). https://paperswithcode.com/method/prioritized-experience-replay