Project 2: Reinforcement Learning - Q-Learning and Deep Q-Learning

Alexander Svarfdal Gudmundsson and Jan Babin

November 1, 2024

Contents

1 Introduction

The main objective of this project is to apply Q-learning and Deep Q-learning to train an agent in a Flappy Bird game. The reason why reinforcement learning is applicable is because we can make an agent run in an simulated environment, even making the agent play at a much faster pace than in real time. In Flappy Bird you as the player play as a bird in a 2d environment where there are two pipes that appear to the right of the screen one coming from above and one coming from the top, with each consecutive frame the pipes move closer to the bird, and your task is to locate the bird by flapping its wings to make it fit between these two pipes. Flappy Bird is an interesting game to take on because it has a good balance between simplicity and challenge. Q-learning algorithm uses a tabular structure to learn every single possible state of the game to then take the best possible action that it knows. The Deep-Q learning uses function approximation to estimate whats the best action depending on what state the game is in.

2 Problem Description

We were tasked with finding a sweetspot for the policy so that it would be as good as possible while involving as little game simulations as possible. For our problem, having unlimited training episodes should (at least for q-learning) result in convergence towards an agent that never dies in the game. However, this is obviously not feasible in reality. Hence, looking for a good-enough solution is very much motivated by how such problems would be tackled in the industry.

3 Characterization of the Environment

The environment techniqually is *continuous* since for example the velocity of the bird or the coordinates of its whereabouts could be represented by float values. In practise, it is probably both *discrete* and *finite* as for the bird's position we can consider a integer grid of pixels and the velocity may only take values from a countable and finite set of float values. Furthermore, the agent may only choose from two possible actions for every state: to flap or not to flap - that is the question.
The game is largely deterministic, with the bird's movements being accurately predictable as well as the pipes' behaviour once they have appeard. However, there is obviously some *stochasticity* involved in their generation: The location of newly spawned pipes is random. Still, one could argue this does not make it stochastic from the agents point of view, but more about that in the following sections.
FlappyBirds is an episodic game: An episode starts with the bird alive and ends once it dies. Starting another iteration is independent of the last one. For an agent, to make informed decisions a state of FlappyBirds should contain the position and velocity of the bird as well as the distance to and the height of the next pipe. Furthermore, the game can inherently be modelled as a Markovian process: Knowing the current state is sufficient to predict the (close) future and choose the next move. That is, if we know the status of our bird and the position of the next pipe, an agent can be able to maneuver without crashing, unknowing of previous pipes or ones that will come after. Knowing subsequent pipes is only interesting for training to evalue future rewards.

4 Implementation of Q-Learning

4.1 State Representation and Action Space

State variables that describe the current environment that the agent is in:

- **player_y**: The current vertical position of the bird
- **next_pipe_top_y**: The horizontal position of the next top pipe
- **next_pipe_bottom_y**: The horizontal position of the next bottom pipe
- **next_pipe_dist_to_player**: The horizontal distance from the next pipe to the bird
- **next_next_pipe_top_y**: The horizontal position of the top pipe after the next
- **next_next_pipe_bottom_y**: The horizontal position of the bottom pipe after the next
- **next_next_pipe_dist_to_player**: The horizontal distance from the pipe to the bird after the next
- **player_vel**: The current velocity of the bird

The actions that the agent can take:

- **flap**: The agent can choose to flap the wings causing the bird to gain altitude
- **no-flap**: The agent can choose to not flap the wings causing the bird to lose altitude

For the Q-learning we opted to bin (discretesize) the values so that the q-table would not be insanely big. We split the space of these attributes (except the velocity) evenly into 15 intervals each, resulting in at most 64125 different states to consider (not all of those are reachable).

4.2 Q-Learning Algorithm

The core of Q-Learning is a table, that assigns a Q-value to each combination of state s and action a. These values represent the expected future rewards for taking that action from that state. Training equals to constantly modifying the table, updating the Q-values. When training, Q-Learning chooses in a state the action that has the highest Q-value (*exploitation*) or a random action (*exploration*), as governed by a hyperparameter ϵ. Then it observes the next state s' and reward r to update the Q-table:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma \max_{a'} Q(s',a') - Q(s,a) \right]$$

5 Implementation of Deep Q-Learning

5.1 Model Setup

Explain the structure of the neural network, including the input normalization, hidden layers, output layer, and activation functions. The action space is the same as above for the Q-learning one but the state representation is not binned in this case but instead

normalised using min-max normalization. The description stated a nice starting point, using two networks with hidden layers, 100 neurons in the first layer and 10 in the second layer. These two networks will then be used in tandem to train the model.

5.2 Deep Q-Learning Algorithm

The algorithm uses the idea of Q-learning to determine Q-values for actions in state but it does not use a Q-table, it uses a deep neural network to approximate the Q-values. The deep neural network is then called a Deep-Q-Network. The algorithm makes the use of a replay buffer that stores past observations, it breaks correlations and stabilizes training by learning from many past experiences. The algorithm also makes use of a target network that gets replaced by the main network on a predefined interval, the traget network provides a stable Q-value to improve stability in training. Furthermore the algorithm uses batch sized sample from the replay buffer to update the Q-network.

6 Experimental Results and Analysis

6.1 Learning Curve Analysis

We employed Learning Curve Analysis to compare Q-Learning and Deep Q-Learning. For this we incrementally trained the models for blocks of 1000 episodes, ran the trained policy / model up to that point on the game and averaged the scores. We then plotted these scores against the number of episodes to get an idea of how the quality of the policy develops over time / training episodes.

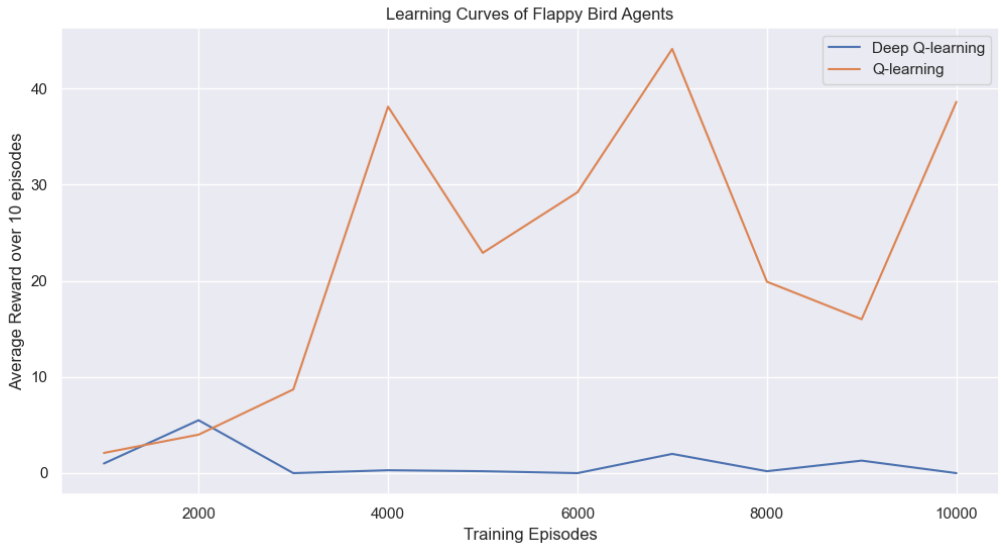

Figure 1: Learning curves for Q-learning and Deep Q-learning agents in Flappy Bird. The average reward over 1 episode is plotted against the number of training episodes.

From this, one can safely assume that the Q-Learning agent will become better as the training amount increases. However, for Deep Q-Learning the same could be expected but this is not indicidated by the plot. Assumingly, we would have to optimise the hyperparameters further. We decided not to do that and stick to Q-Learning for hyperparameter

optimisation, so as to cater to time constraints we had. Q-Learning is quicker to train when only accessing CPU-power.

6.2 Parameter Tuning

- **Discount rate**: Determines how far ahead we look into the future in terms of the rewards i.e. how farsighted or nearsighted our agent is
- **Learning rate**: Hyperparameter that controls how much value the agent treats our new information in contrast to the old Q-value in the table
- **Greedy epsilon**: This controls how much we exploit vs explore, can be from 0 to 1, the higher the epsilon the more we explore

For the hyperparameter tuning, we use random grid search to find the best combination of the parameters for the Deep Q-Learning agent. We randomly tried 20 different hyperparameter combinations and trained each of the corresponding models on 2000 episodes.

6.3 Best Performing Agent

Table 1: Best Hyperparameter Combination for Q-learning

Hyperparameter	Value
Discount Rate (γ)	0.950
Learning Rate (α)	0.096
Exploration Rate (ϵ)	0.014

The best combination of hyperparameters we could find is shown in Table 1. We then initialized a new agent with these settings and also an agent with default settings and trained both for 3000 episodes. The tuned agent performed significantly better (see Figure **??**). After that, we trained our best agent for 10000 episodes, tracking the progress by plotting the learning curve as can be seen in Figure **??**.

7 Conclusion and Future work

Q-Learning turned out to be a robust method to find a reliable agent as was expected. Using 15 buckets for the state attributes is precise enough to control the bird, so that it only dies in "unforeseen" states, that is, states that the agent had not encountered that often. We assume that our agent will converge towards a perfect player of the game (i.e. bird never dies) as the number of episodes increase.
The Achilles' heel of our solution is that the Deep Q-learning agent did not perform as well as we hoped, after certain amount of episodes it converged to be a worse agent, this pattern can be seen in figure 1. Due to this shortcoming the future work would consist of fixing this agent so it does not get worse over time. As figure 1 depicts the Deep Q-learning agent shows some improvement trianing on only 2000 episodes but then it plumits after 3000 episodes to a average reward over 1 episode of around 0. As of now we dont know the reason why, we tried to troublshooting this (possibly spent too much time on that) but with no luck.

Figure 2: Created by Google Gemini