Háskólinn í Reykjavík

Machine Learning

Project 2 Report

Group 10

Björn Víkingur Borg
Kt: 160902-2050
Jóhann Ágúst Hafsteinsson
Kt: 041002-2630

Teacher: Christoph Lohrman

Introduction

In this project, we go into reinforcement learning by creating an agent to play Flappy Bird, a game where players guide a bird through pipes without crashing. We started with a simple Q-learning algorithm, which helped the agent learn from key states like the bird's position and the proximity of pipes.

As we advanced with the project, we moved on to Deep Q-learning, using neural networks to better manage the game's complexities and improve the agent's performance. Throughout our experiments we explored different hyperparameters and strategies to see how they affected the agent's effectiveness. Our goal for this project was not only to develop a skilled Flappy Bird player but also to get a better understanding of reinforcement learning principles in practise.

Environment

The Flappy Bird environment is a discrete and stochastic system, meaning that it has a finite set of actions, specifically the bird can either flap or do nothing, while the outcomes of those actions can be uncertain, like when the bird collides with a pipe due to varying positions. Each game session is episodic, starting when the game begins and ending when the bird crashes. The state of the environment is defined by several key variables: the bird's vertical position, the heights of the upcoming pipes, the distance to the next pipe, and the bird's velocity. This information captures the current situation well enough to satisfy the Markov property, where the future state relies on the present state and not the history of previous states.

With the dynamics of the environments, we don't really know the exact rules for how the game changes states but we can figure it out by playing and learning.

When it comes to figuring out how the bird should act, we could technically use a tabular approach like regular Q-learning if we break down the state into simpler pieces. But since the bird's situation involves lots of continuous variables, that could get messy really quickly. That is why deep Q-learning could be more beneficial to us as it uses neural networks to handle all those continuous inputs and helps the agent learn better by generalizing from similar situations. It also makes the learning process smoother with things like experience replay and target networks. So deep Q-learning is better suited for Flappy Bird than a simple table would be.

Evaluate Q-learning

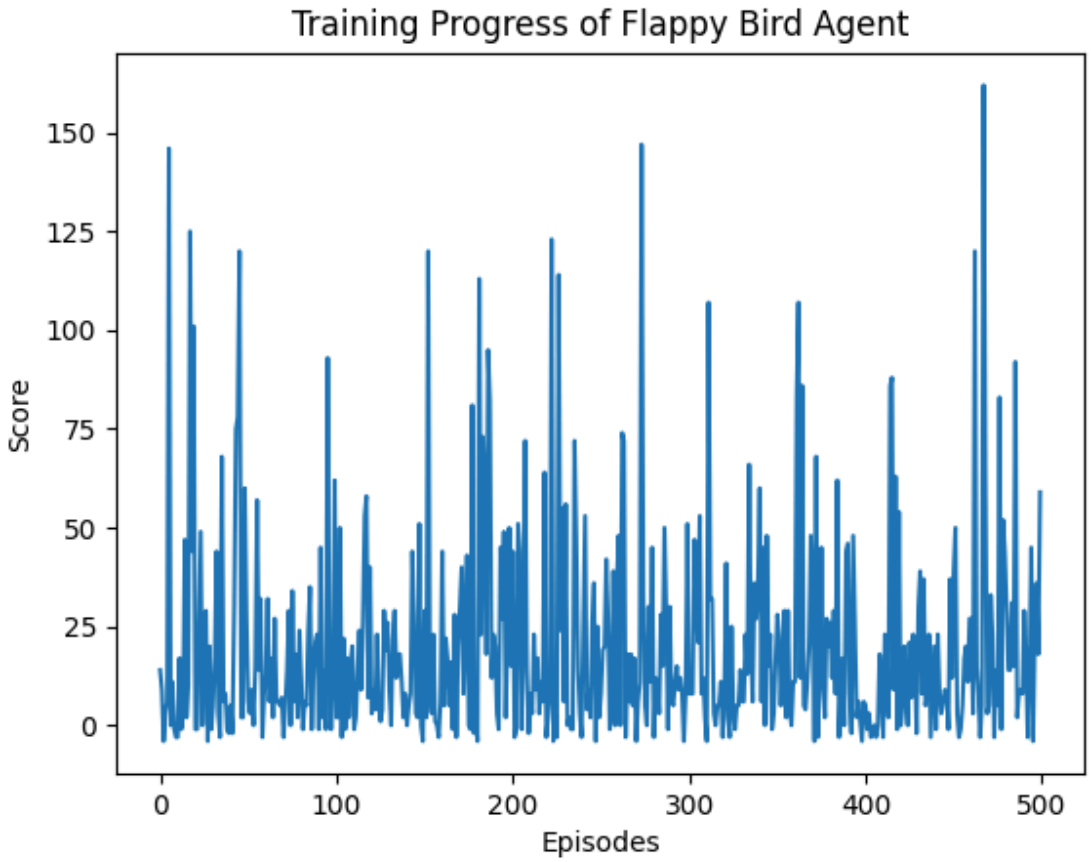
We started off our project by implementing Flappy Bird using a simple Q-learning algorithm. The idea was to teach our bird how to navigate through the pipes without crashing.

In our implementation, we used a table to store Q-values for different actions the agent could take in various states. The states included important information like the bird's current height, the height of the next pipe, the distance to that pipe, and the bird's current velocity. We decided to break these continuous values into 15 intervals, allowing us to manage around 64.125 possible states, although not all of these are realistically reachable in gameplay.

To encourage the agent to explore different actions, we used an epsilon-greedy strategy. This meant that with a probability of 0,1, the agent would explore a random action instead of always going for the best-known action. The learning rate was set to 0,1 so the agent would make significant adjustments to its Q-values based on new experiences. We set the reward structure according to the assignment description to give one point for passing through a pipe, 0 points for each flap and -5 points for crashing, which helped shape the agent's learning process.

During training, we observed the agent playing the game, collecting experiences and updating its Q-values accordingly. After training for a certain number of episodes, we plotted the scores to see how well the agent was learning over time. The results showed us how the agent improved as it gained experience and adjusted its strategy based on the rewards it received.

Here is an image after running the basic Q-learning algorithm for 500 episodes:



Looking at the graph, there is a lot going on. The scores jump up and down quite a bit, which means the agent has had a lot of ups and downs while learning. Sometimes it scores really high, even hitting 150, but there are also a lot of low scores, indicating it crashed at the beginning of the program often (we know that because of the reward system). Overall there seems to be a slight upward trend as the episodes go on, suggesting that the agent is getting better at playing the game over time. Even though there are a whole lot of fluctuations, it looks like the training is heading in the right direction that can be made better with some adjustments.

Deep Q-learning

After implementing the basic Q-learning algorithm, we moved on to Deep Q-learning to enhance our agent's performance. The main reason for moving on to deep Q-learning was to be able to better deal with larger state spaces and leverage the power of neural networks to approximate the Q-values instead of storing them in a table.

In our DeepQAgent class, we set up two neural networks, one for estimating the Q-values and another one for serving as a target network. This setup helps stabilize the learning process by decoupling the Q-value updates from the target values. We initialized both networks with a specific architecture that includes two hidden layers which allows the network to understand better how the bird's position and the pipes location relate to the best action to take.

To train the agent, we implemented experience replay, which means that we store past experiences and sample from them during training. This method allows the agent to learn from a diverse set of experiences rather than only the recent ones.

Also, we implemented an epsilon-greedy strategy for exploration, like we did in Q-learning, but with some tweaks. As the agent trains it starts with a higher exploration rate and gradually decreases it, allowing it to exploit what it has learned over time while still exploring new strategies early on.

Overall the switch to Deep Q-learning provided our Flappy Bird agent with a more robust learning mechanism which enables it to perform better and navigate the pipes more effectively.

Making the best agent

References to Images can be seen below.

We experimented with three epsilon decay values (0.999, 0.998, and 0.975) to see how different rates of exploration affected the agent's learning. The epsilon decay rate controls how quickly the agent shifts from exploring (trying out new actions) to exploiting (sticking to what it has learned as "best" actions).

A higher decay value slows down this shift, allowing the agent to explore more over a longer period. In Image 2, with an epsilon decay of 0.998, the agent still explored a fair amount but gradually started committing to certain actions. In Image 4, with a slightly higher decay of 0.999, the agent held onto exploration even longer, which helped prevent it from locking into suboptimal strategies too quickly. The result was slightly more stable performance, as the scores in Image 4 show fewer extreme highs and lows compared to Image 2. The ranking difference isn't huge, and the scores are still quite random overall.

This suggests that while a slower epsilon decay helps the agent keep exploring and avoid bad habits early on, it's not enough on its own to produce consistently good performance. The agent likely needs even more training time to take full advantage of this slower decay rate and further stabilize its strategy

We tried different batch sizes across the images—10 in Image 1, 20 in Image 2, and 50 in Image 4. A larger batch size means the agent learns from more experiences at each step, which theoretically could help it generalize better.
With the larger batch size of 50 in image 4, the scores seem a bit more controlled and less prone to random spikes compared to the smaller batch sizes. This makes sense as a larger batch size should help average out the noise in the agent's experience to get a smoother experience. The scores still look fairly inconsistent in images 1 and 2 but Image 4 suggests that a higher batch size helps generalize better.

Target Update Interval:
In image 1, we set the target update interval to 20, which meant the agent was updating its target network pretty often. Then, in image 4, we decided to increase to 70. The idea behind that was to give the agent more time to settle down before it made those updates.
Looking at the results, it seems like that higher interval of 70 really helped with stability. With updates happening less frequently, the agent had a more consistent target to aim

for, which helped reduce some of the score fluctuations we saw in the earlier images. The difference can be seen in image 4, where the scores are less likely to drop sharply compared to image 1.
Still, its not perfect. There is definitely some score variability left, which suggests that even with a higher update interval, it might not be enough to smooth out the learning completely. But its a step in the right direction.

Image 1:

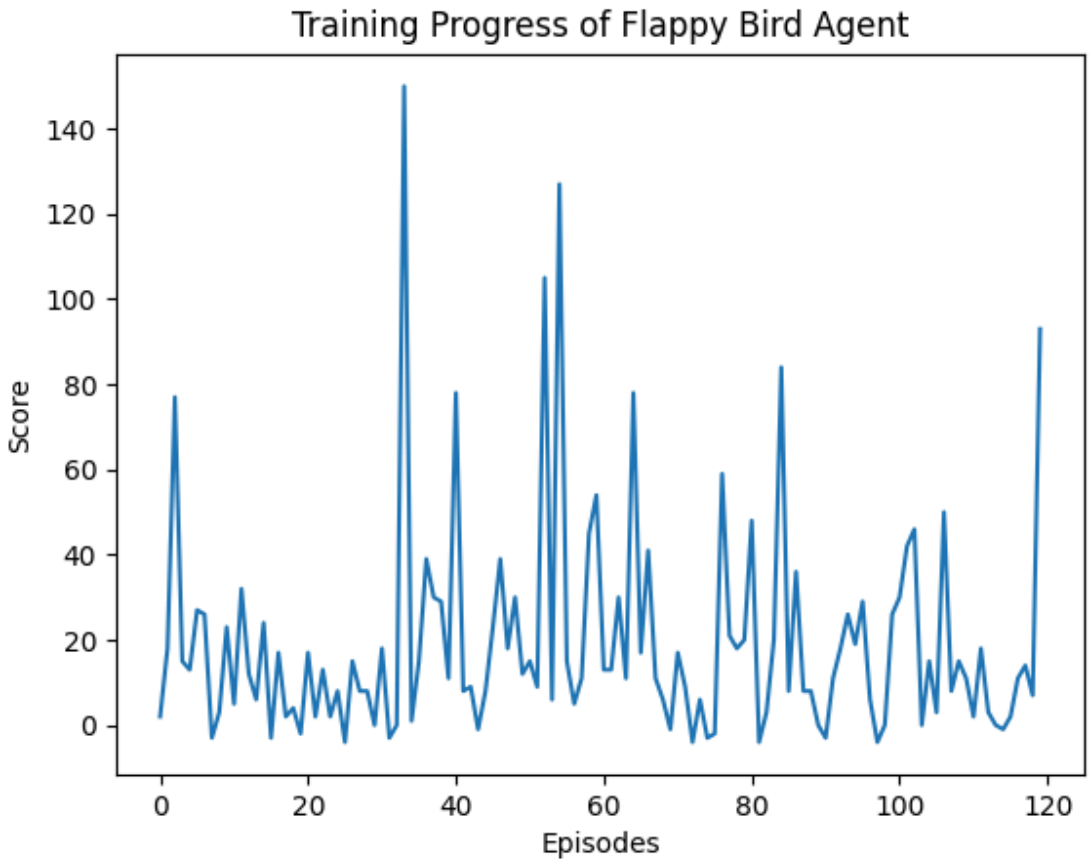


Image 2:

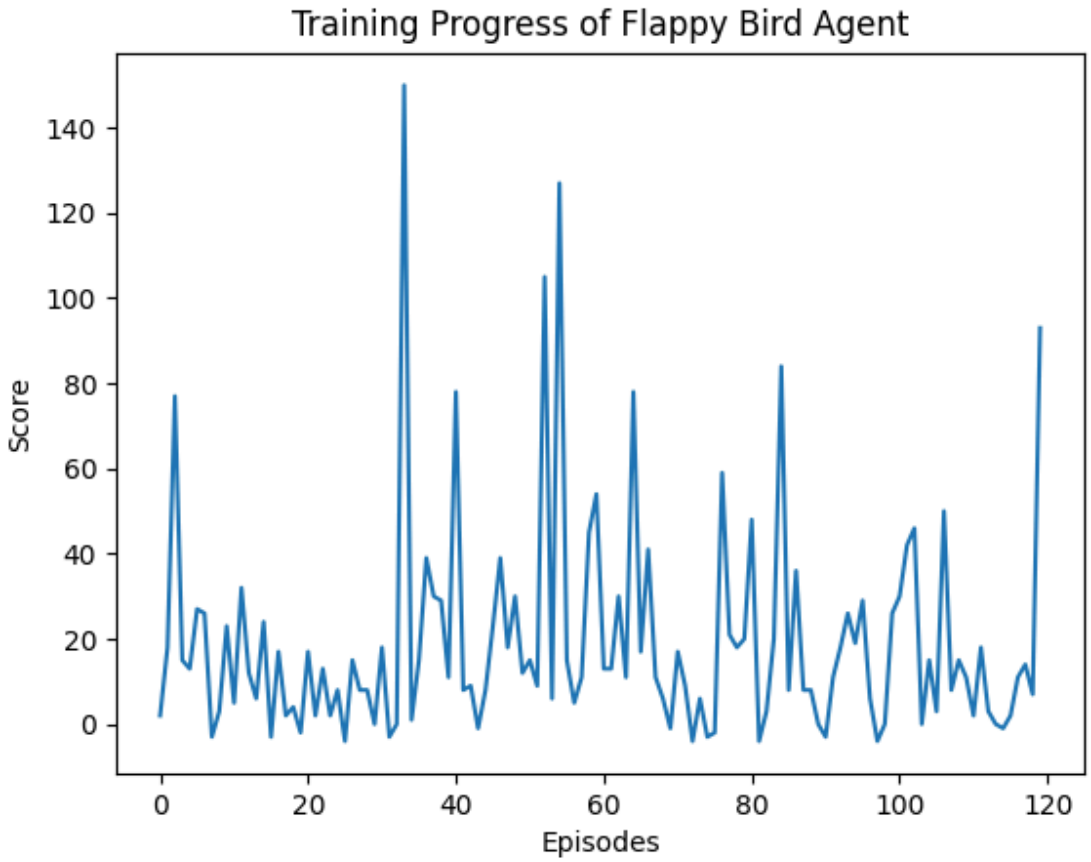

Training Progress of Flappy Bird Agent
Score
Episodes
0
20
40
60
80
100
120
140

Image 3:

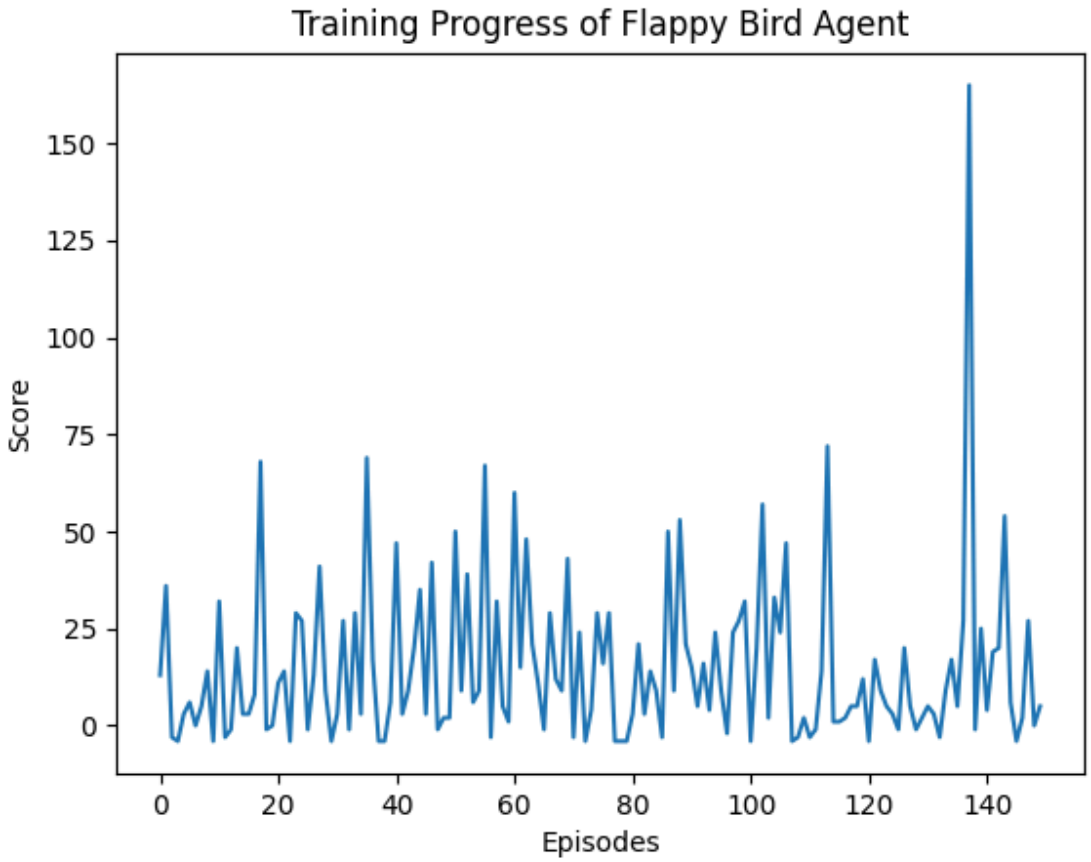
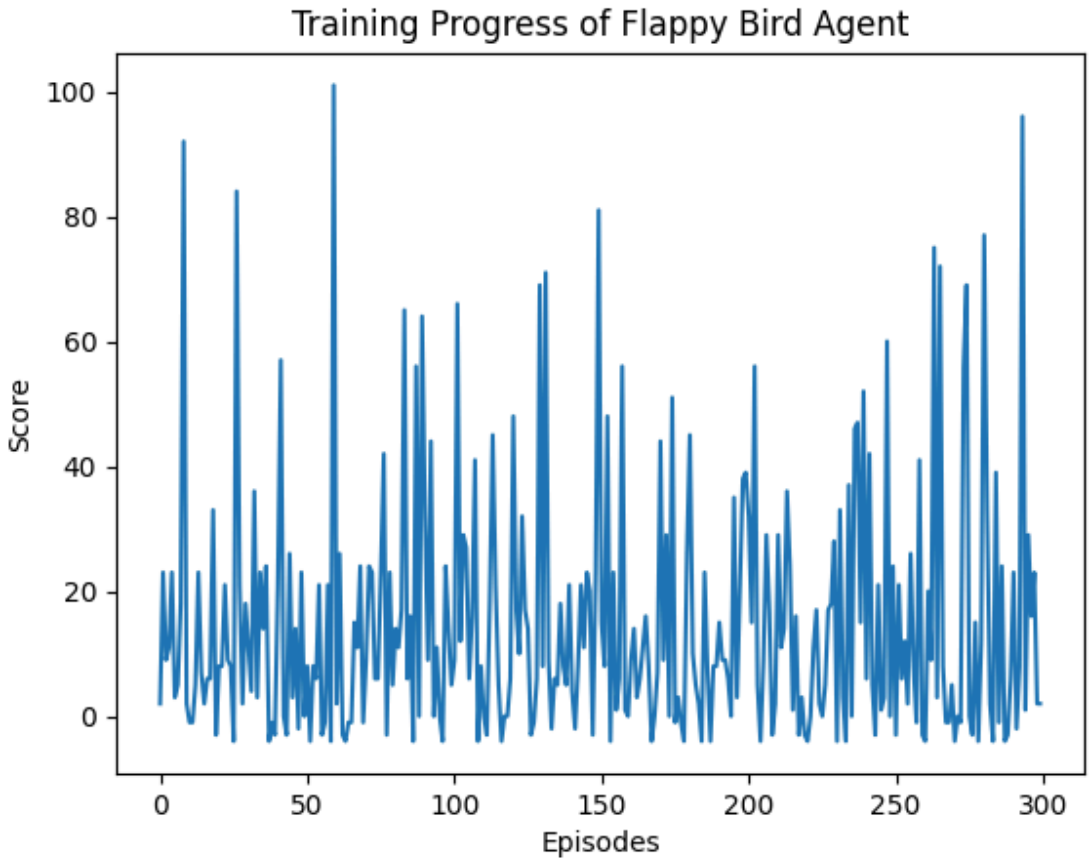

Training Progress of Flappy Bird Agent
Score
Episodes
0
25
50
75
100
125
150
0
20
40
60
80
100
120
140

Image 4:



Conclusion on the Best Parameters for Our Agent

After trying out different settings, here's what we found worked best:
Epsilon Decay: Slower epsilon decay (like 0.999) gave the agent more time to explore instead of jumping to conclusions too fast. This helped it avoid bad habits early on, leading to smoother scores over time. So, letting it explore a bit longer was definitely a good move.

Batch Size: A bigger batch size (like 50) helped the agent learn from more experiences at once, which smoothed out the ups and downs. With a smaller batch size, the scores were all over the place, but with 50, the learning felt a lot steadier.

Target Update Interval: Updating the target less often (setting it to 70) seemed to keep things stable. The agent wasn't constantly adjusting to a moving target, which helped avoid big score jumps and dips.

Overall, the combination of a slower epsilon decay (0.999), a bigger batch size (50), and a higher target update interval (70) seemed to make the agent's learning more consistent and less chaotic.