Teaching an agent to play flappy bird with q-learning

Faculty of Computer Science and Engineering,

Skopje

Andrej Slavejkov

196047

Abstract

In this work we will be looking at a method for training an artificial intelligence agent to play the popular game, Flappy Bird, using a technique called Q-learning.In this game, the player controls a bird and must guide it through a series of obstacles without hitting them. The agent is rewarded with a score of 1 for each pipe that it stays passes through, but gets a negative reward of -1000 if it dies.

This approach focuses on teaching the agent to make decisions based on its past experiences and the potential rewards it can earn by taking different actions. By using Q-learning, we can gradually improve the agent's performance over time through trial and error, leading to better gameplay and higher scores. The results of these experiments demonstrate that this approach is effective for teaching an agent to play Flappy Bird, and can be applied to other games and have real-world applications.

Introduction

Flappy Bird is a classic and highly addictive video game that has captured the attention of millions of players worldwide. The game involves guiding a bird through a series of obstacles, and players must navigate through the gaps without colliding with them. While the game may appear simple, it’s actually quite challenging and requires quick reflexes and precise timing.

In this paper, I explore the use of Q-learning to train an artificial intelligence agent to play Flappy Bird. The game is implemented in Python using the Pygame library, and the Q-learning algorithm is implemented using a handmade code. The code is developed by referring to other sources[1][2][3] and incorporating various modifications to suit my specific requirements.

My approach focuses on creating an environment where the agent can learn to make decisions based on its past experiences and the potential rewards it can earn by taking

different actions. The goal is to train the agent to maximize the score by staying alive as long as possible and avoiding collisions with obstacles.

Related Work

Previous research has shown the potential of using artificial intelligence and machine learning techniques to train agents to play video games. A well known example is DeepMind's AlphaGo[4], which used a combination of deep reinforcement learning and Monte Carlo tree search algorithms to beat the world champion in Go.

In the context of Flappy Bird, there have been several studies that have explored the use of machine learning algorithms to train agents to play the game. One notable example is the study by Chen et al. (2018)[5], who used a variant of the Q-learning algorithm, called Double Q-learning, to train an agent to play Flappy Bird. They also used a gated recurrent unit to model the game's state and improve the agent's decision-making process. Their experiments showed that their approach outperformed traditional Q-learning and achieved higher scores on the game.

Another example is the study by Cao et al. (2019)[6], who used a variant of the deep Q-learning algorithm, called DQN, to train an agent to play Flappy Bird. They used a convolutional neural network to extract features from the game's state and make decisions based on them. Their experiments showed that their approach achieved high scores on the game, even in the presence of noise and distractions.

My approach takes inspiration from the previous studies by using a handmade implementation of the Q-learning algorithm and Pygame library to train an agent to play Flappy Bird.

Environment Preparation

Preparing the environment for the Flappy Bird game was a crucial step in my implementation of the Q-learning algorithm. I chose to use the Pygame library, which is a popular game development framework for Python. The first step was to set up the environment with the necessary game elements, such as the background, bird, pipes, and score display.

I referenced various sources to guide the general structure of the game. These sources included Pygame tutorials, Flappy Bird game clones, and online forums where game developers shared their knowledge. I modified and combined the code from these sources to create my own implementation of Flappy Bird, tailored for use with the Q-learning algorithm.

Agent, methods and experimentation descriptions

In this case there is only a single agent which plays flappy bird. At the start of the game the environment is initialized and we set the agent's initial position. From there the game enters a loop where it continuously updates the game state, allowing the agent to make decisions at each step. The agent gets information about the bird's position, the positions of the pipes and the speed at which they are coming towards the agent. The agent then passes this information through the AI model, which then decides whether or not the agent should take one of the two available actions to it, to jump or not. If the agent jumps the bird velocity is updated to simulate the jumping and if it stays in place the velocity gets updated to simulate gravity.

The method we use to teach the agent what to do is by saving each action it takes into a q-table along with the reward that action earned. It then uses that information to calculate the best action next time it is in the same position. The agent is rewarded 1 point every time it passes a pair of pipes, but gets a -1000 point penalty whenever it collides with a pipe or the ground. This reward structure incentivizes the agent to avoid collision with the pipes and

maximize its score. At the end of each game the agent updates its knowledge through updating the expected reward for each state-action pair. It then uses this knowledge to play more effectively in the next iteration.

Normally one way to experiment on a q-learning model is by changing epsilon, the parameter which determines whether the agent will be more prone to explore new more random actions or stick to what it has already learned. This parameter is then usually set to 0 for the testing stage so that the agent uses what it's learned. In this experiment the agent has no such parameter as I didn't believe it is necessairy for a game like flappy-bird.

The way I will be experimenting with this agent model is by changing the gamma value, or what is the equivalent in my code as I don't directly use one.

```
def Q_update(x_prev, y_prev, jump, reward, x_new, y_new):
    if jump:
        Q[x_prev][y_prev][1] = 0.9 * Q[x_prev][y_prev][1] + 0.1 * (
                    reward + max(Q[x_new][y_new][0], Q[x_new][y_new][1]))
    else:
        Q[x_prev][y_prev][0] = 0.9 * Q[x_prev][y_prev][0] + 0.1 * (
                    reward + max(Q[x_new][y_new][0], Q[x_new][y_new][1]))
```

This is my formula for calculating the q-table depending on the x and y position of the agent as well as the action it has taken. Here, 0.9 is the discount factor, which determines the importance of future rewards, and 0.1 is the learning rate, which determines how much the new information should be weighted compared to the previous information in the Q-value. The max function calculates the maximum Q-value for the next state.

Results

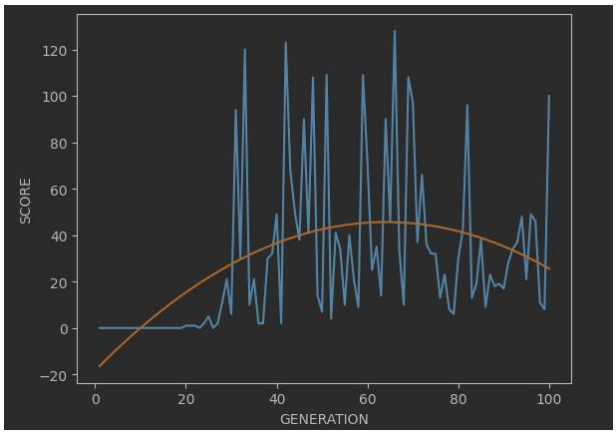
Results of the Gamma Change Experiment:

In this experiment, I explored the effect of changing the discount factor gamma on the performance of the agent. I tested four different gamma values: 0.3, 0.6, 0.9, and 0.99, and for each gamma value, I ran 100 episodes of the game. I recorded the total reward accumulated by the agent in each episode and calculated the average reward for each gamma value.

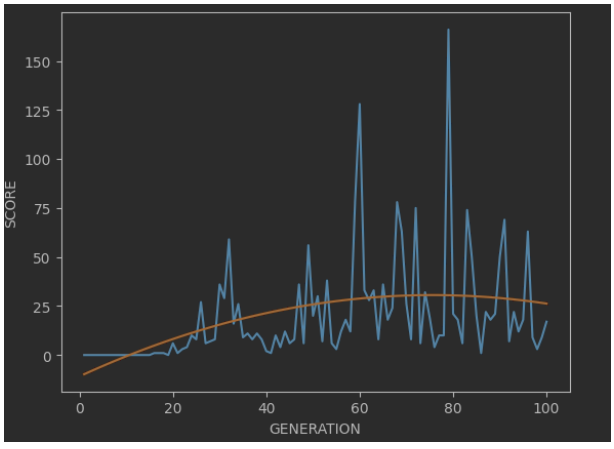
The results were as follows:

(the graphs are representing the score the agent gets in each of its generations. The curve is a hyperbolic mean of the score)

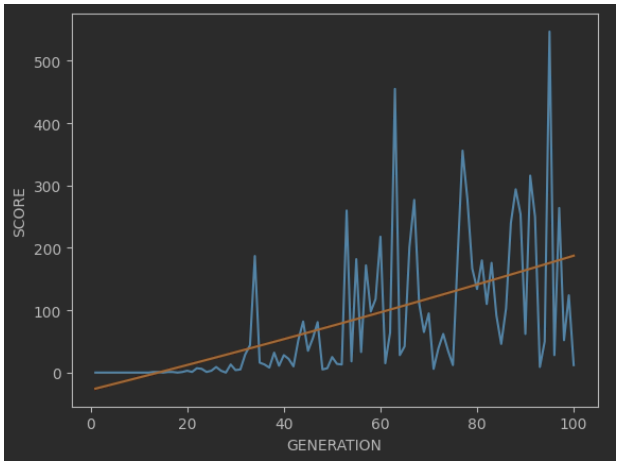
$\gamma = 0.99$, average score: 29.75



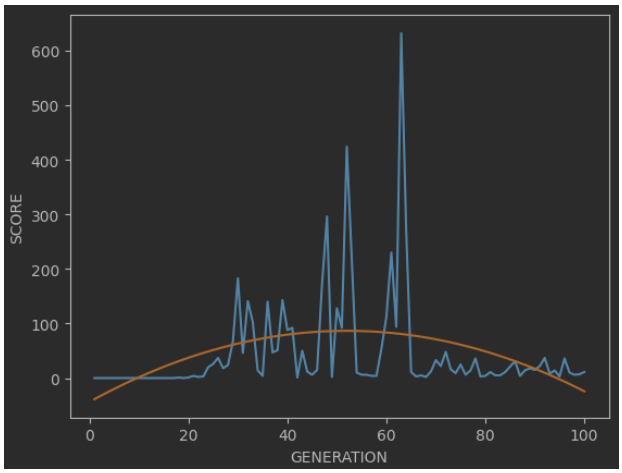
$\gamma = 0.9$, average score: 19.96



γ = 0.6, average score: 77.69



γ = 0.3, average score: 46.35



We can clearly see the differences that different gamma values bring about. When the discount rate is too high the model isn't learning quickly enough while when its too low the higher learning rate seems to overshoot so that it even when it learns something and does it well for a few generations, the agent quickly loses its grasp on the correct actions that it should have been taking. The best results for a set of 100 runs seems to be 0.6 discount rate and 0.4 learning rate. This way the agent learns fast enough to learn in the limited amount of runs it has but not too fast so that it can't commit the knowledge it has gained before its overwritten.

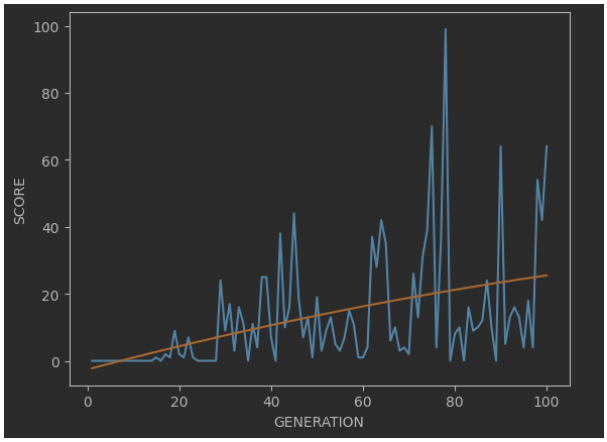
Results of the Reward Change Experiment:

In this experiment, I examined the effect of changing the rewards on the performance of the agent. I tested two different reward settings: the first setting, in which the reward for

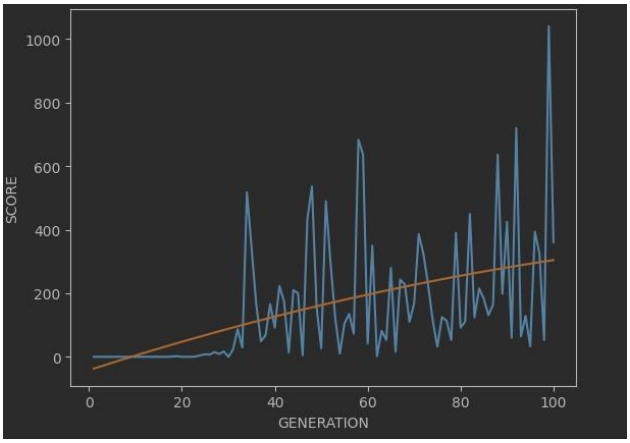
successfully jumping over an obstacle was increased, and the second setting, in which the reward for hitting an obstacle was decreased. I ran 100 episodes of the game for each reward setting and recorded the total reward accumulated by the agent in each episode. The learning and gamma rate used for this experiment will be 0.6 and 0.4 respectively as they were most effective in the previous one.

These are the averages after 3 runs of 100 episodes with the reward increased to 10:

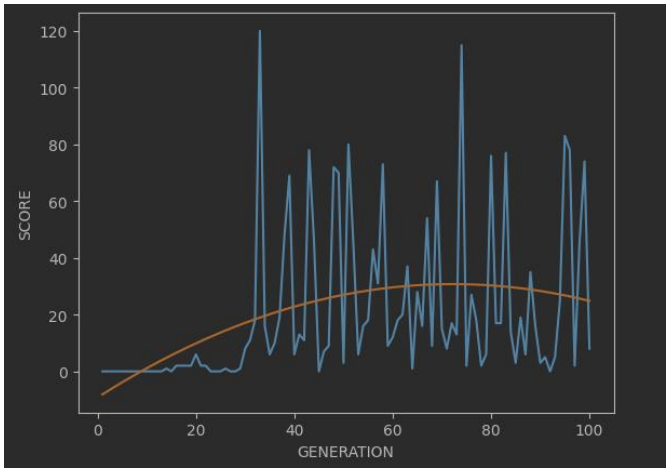
12.96



153.8,



20.76



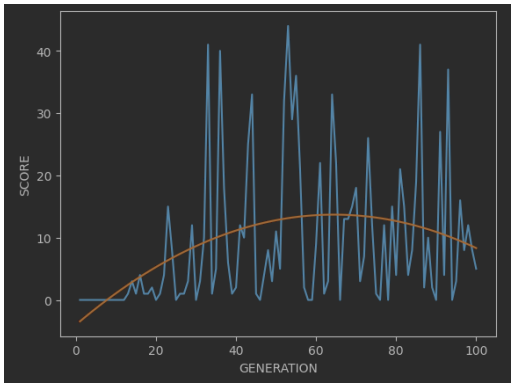
From these results we can see that increasing the reward for surviving seems to destabilize the performance of the agent where if it stumbles onto a good pattern by luck it will more quickly learn how to reproduce it however, it becomes slower and more inconsistent and learning without this luck of hitting the right combination.

When looking at the results from decreasing the punishment for dying, I experimented with lessening the punishment to -100, -300 and -500. I will also be using the -1000 standard as a benchmark.

Results:

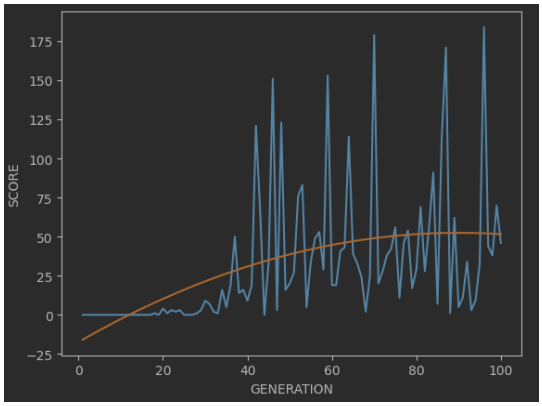
Dying reward: -100

Average score: 9.32



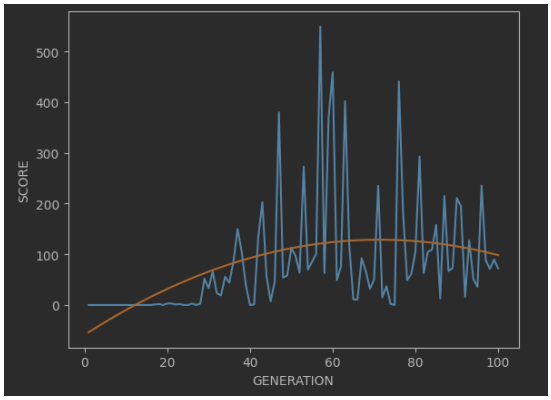
Dying reward: -300

Average score: 31.82



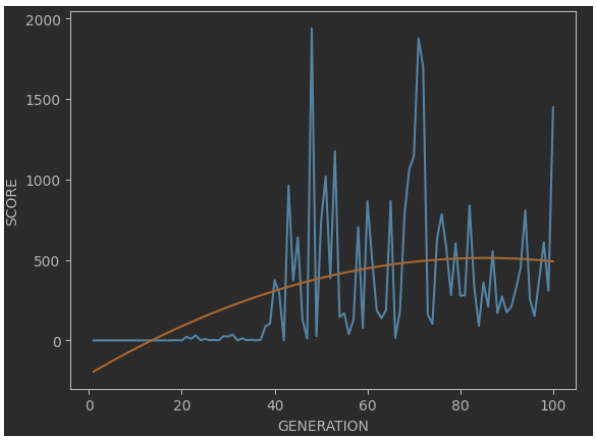
Dying reward: -500

Average score: 82.01



Dying reward: -1000

Average score: 308.76



For -1000 reward on dying I reran the tests again and as expected it remains the best way to train the agent. These results clearly present that in a game like Flappy-bird it is important to heavily punish the agent for making a wrong decision as one wrong decision means losing the game.

Conclusion

In conclusion, the experiments conducted in this paper have shown the impact of gamma and reward on the performance of the q-learning algorithm in the Flappy Bird game. The results indicate that the choice of gamma and reward values has a significant effect on the agent's ability to navigate and take actions in the game successfully. Specifically, I found that a higher gamma value leads to better long-term planning, while a lower gamma value favors more immediate rewards. Additionally, increasing the reward for successfully navigating the game leads to a more unstable performance. This suggests that further research can be done to explore the optimal values of gamma and reward for Flappy-bird game and similar environments. Overall, this study demonstrates the importance of careful parameter tuning in reinforcement learning algorithms and highlights the potential for improving agent performance in complex environments by adjusting reward and discounting factors

References

[1] https://github.com/Talendar/flappy-bird-gym

[2] https://github.com/kyokin78/rl-flappybird

[3] https://github.com/yenchenlin/DeepLearningFlappyBird

[4] Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., ... & Hassabis, D. (2016). Mastering the game of Go with deep neural networks and tree search. Nature, 529(7587), 484-489.

[5] Chen, T. H., Liang, C. Y., Lee, T. Y., & Huang, C. M. (2018). Deep reinforcement learning for flappy bird using gated recurrent unit and double q-learning. Journal of Information Science and Engineering, 34(2), 323-335.

[6] Cao, Y., Chen, X., & Wang, S. (2019). Deep Q-learning for flappy bird with convolutional neural networks. In 2019 IEEE 6th International Conference on Cloud Computing and Intelligence Systems (CCIS) (pp. 557-561). IEEE.