Training Flappy Bird with Deep Reinforcement Learning

Dimitar Georgiev

Ss. Cyril and Methodius University Skopje

Faculty of Computer Science and Engineering

dimitar.georgiev@students.finki.ukim.mk

***Abstract* - Flappy bird was one of the most popular arcade mobile games that ever existed. Its simplicity and limited actions make it a perfect environment to experiment on with various reinforcement learning techniques. Although this game is different from other arcade games that are frequently used for reinforcement learning (for example it does not have a winning ending state), this paper demonstrates how the environment can be used to train the agent (bird) to achieve the highest score it can while trying not to fall on the ground or hit any obstacles. The paper explores using multiple reinforcement learning architectures in combination with two versions of the environment and multiple different reward systems.**

***Keywords* – reinforcement learning, deep learning, DQN, DuelingDQN, Flappy Bird**

I. INTRODUCTION

Reinforcement learning as a tool is becoming more and more relevant in many scientific fields because it is suitable for solving many modern problems that have more than one correct solution. Combined with other machine learning techniques and environments it can produce optimal solutions for problems or strategies and approaches people have not even thought of. Of all machine learning paradigms, reinforcement learning most closely resembles the learning of living beings as it utilizes exploring, experimenting, trial and error and having consequences of certain actions (rewards and punishments). A problem, in order to be solved with reinforcement learning, is modeled through agents and environments. An agent is an entity which interacts with the environment and responds to it while exploring and searching for the optimal solution, all in absence of a supervisor. The optimal behavior is learned by using a reward system which punishes and rewards the agent, based on its decisions (the agent tries to maximize the reward). Although the agent explores and learns to behave in a given environment, its actions can prove useful in similar unseen situations.

Flappy Bird was a mobile arcade game developed by Gears studio. It came out in 2013 and within a year it ended up as the most downloaded App on the App Store [1] thanks to its addictive nature. Despite its huge success (the game was earning around $50 000 a day through advertising) it was discontinued in 2014 for unknown reasons. Since then, many similar games were developed, some of them nearly identical, as an attempt to recreate the game. The game consists of a bird called Faby (controlled by the player) which moves from left to right. The task of the bird is to stay alive as long as it can by avoiding falling to the ground or hitting the obstacles. The obstacles are vertical pipes, one above the other, with a gap between them, so the only way to pass through the pipes is to fly through the gaps. Since the bird cannot fly by itself, the player has to control it by touching the screen whenever the bird has to flap its wings and dash upwards in the air (while simultaneously moving right). When not flapping, the bird starts gradually falling down. The player gets 1 point when the bird passes through a pair of pipes.

The simplicity of Flappy Bird makes it an ideal reinforcement learning environment. The agent is the bird itself, and it has only two actions – flap or do not flap. The challenge is to learn exactly when to flap in order to adjust the height in time to pass the gaps in the pipes (this is tricky even for a human). Each new pair of pipes has a differently placed gap (random vertical position) which makes the environment different every time it starts. Most arcade games that are used as environments for reinforcement learning have both winning and losing terminating states, which either punish the agent a lot of points (losing state) or reward the agent with the biggest defined reward (winning state). Flappy bird is different in terms of terminating states, since it only has a losing state (falling down or hitting an obstacle). This is not a problem in the learning process itself, but it makes the goal of the agent to reach the highest score possible instead of winning the game.

II. RELATED WORK

In order to start experimenting with reinforcement learning, I had to find a suitable environment. There are many versions of Flappy Bird implemented using PyGame (a framework in Python for developing games) which are easy to integrate with OpenAI Gym (the library which I used for reinforcement learning in this project). One environment I considered is FlapPyBird [2] which is available on Github. It is a custom implementation of the game using PyGame and is used in multiple similar projects. Another project FlapPyBird-RL by Anthony Li [3] uses this environment for reinforcement learning and it demonstrates a successful result (using q-learning with experience replay). The only downside is that the environment is not integrated with OpenAI Gym and uses a custom framework for performing actions. The source code of this project is available on Github and there is also an article that I found useful on Medium which goes through the whole process.

Another environment I found is DeepLearningFlappyBird [4] (also available on Github) which has another implementation of the game using PyGame, but similarly to the previously mentioned environment it uses a custom framework instead of the gym library. The project explores using convolutional neural networks with a deep q-learning algorithm to train the agent.

The environment I thought was best at the moment (and ended up using in this project) is Flappy Bird for OpenAI Gym [5] which uses the PyGame implementation of FlapPyBird [2] integrated with the gym library. The environment is also available as a Python package and can be installed using pip (flappy-bird-gym). For these reasons it is easy to acquire and use. I will describe this environment in further detail in the Environment section of this paper.

In his student report, Kevin Chan [6] explores one way of using Flappy Bird with reinforcement learning. He uses images as representations of the state of the game, which are transformed to grayscale before being used for learning. In terms of the deep reinforcement learning method, Chan uses convolutional neural networks and experience replay. This report demonstrates successful results but uses a different environment than the one selected in this paper.

III. METHOD

In this section I will describe the environment in detail, the methods used for training the agents and the reward systems I used.

A. Environment

The environment I selected [5] is a very accurate representation of the game Flappy Bird. As mentioned above, it is implemented using the PyGame library and is integrated with OpenAI

Gym (it overloads all the necessary methods used for reinforcement learning). The environment generates evenly spaced obstacles (pipes) with randomly placed gaps (random vertical position). This makes the environment rely on a random factor and produce different settings every time it is started (which makes flying a challenge for the agent).

The starting state of the environment always spawns the agent/bird on the same position – approximately in the middle of the screen and relatively distanced from the start of the obstacles. After the first action is performed (it triggers the start of the game), the agent starts falling down (the environment simulates gravitation) and moving right constantly.

The agent can perform two actions:

a) Flap (this action makes the bird flap with the wings and dash upwards while simultaneously moving right)
b) Do nothing (this action does not make the bird do anything and it simply continues falling down while moving right as the same time)

The lack of actions makes it difficult for the agent to navigate through the air precisely enough to miss all the obstacles. One flap makes a significant change of the agent's height position and it has to be precisely planned when to use this action in order to fly through the gap. Another problem is the relatively short distance between two obstacles which not always gives the agent time to adjust its height before hitting the obstacle.

There are two types of ending states of the game. One state that ends the game is the one in which the agent hits the ground, and another ending state is that in which the agent hits an obstacle (misses a gap). Both of these terminating states are losing states and there are not any winning states in the game. Contrary to many other arcade games, a certain goal which has to be reached in order to win the game does not exist. The only goal is to try to stay alive (not lose) for as long as possible and score as much as you can. The total score increases for one point every time the agent flies through a gap of an obstacle (not to be confused with the reward system which works differently).

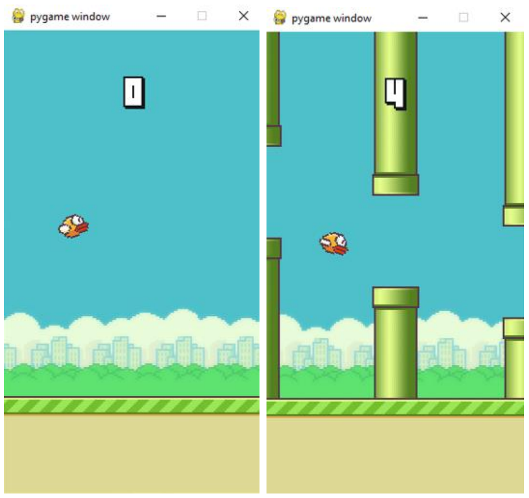


Figure 1: Starting state of the game (left) and the game once the agent reaches the obstacles (right).

The environment has two versions which differ in the way they represent a state. The first version called FlappyBird-rgb-v0 represents each state as an RGB image. The other version FlappyBird-v0 yields some form of feature extraction of the overall state. Instead of an RGB array it returns a vector of two values – the horizontal distance to the next obstacle and the vertical difference between the agent's vertical position and the next gap's vertical position. The former version uses a more descriptive representation of the state but requires more resources to train the agent (neural models with many parameters and performing feature extraction), while the latter is less descriptive at the price of requiring less computational power for reinforcement learning.

B. Reward Systems

The environment itself has an in-built reward system that is independent of the scoring system of the game. Instead of rewarding the agent with one point whenever it passes through an obstacle, it rewards the agent one point with every step taken that doesn't terminate the game. That way, the agent can learn to stay alive as much as possible and figure out by itself how to pass the obstacles. This reward system does not have any negative rewards (punishments) or any other types of rewards, but nonetheless it encourages the agent to learn.

After experimenting with this reward system, I figured I had to tweak it in order to achieve better results. In my opinion the former way of rewarding the agent is not informative enough on where the agent makes either good or bad decisions. My idea was to incorporate some kind of relation between the position of the agent and the position of the next obstacle into the rewards so the agent can learn to stick closer to the height of the gap and not miss it. The most simple way to do this is to add the negative value of the vertical distance between the agent and the middle of the gap to the one point that is given as a reward. Since the above-mentioned distance is a small decimal number between 1 and 0 (very close to 0 in most cases) I was not sure if it would affect the learning process significantly, so I implemented another version which works the same but multiplies the negative distance by 10 to amplify its impact.

Another thing I thought was going to improve the learning process was introducing a losing punishment to the reward system. Thus, I made a new reward system in which a punishment of -10 (or -1 in other attempts) is given to the agent when it makes an action that terminates the game. This change in the reward system proved useful as it encouraged the agent to learn more smoothly from its experiences that ended the game, although it did not make a very big impact in the final results.

C. Used Architectures

In combination with the two different versions of the environment and the reward systems described above, I experimented with a few deep reinforcement learning architectures and compared their performances.

Deep Q-Network (or DQN) is the first architecture for reinforcement learning proposed by DeepMind [7] that utilizes deep learning. The Deep Q-Networks overcomes unstable learning by using 3 important techniques: experience replay, target network and reward clipping. Experience replay refers to storing past experiences (states, transitions, rewards and actions) in the agent’s memory, in order to take samples as mini-batches for training purposes. This kind of sampling is used in order to produce various experiences for the agent (while the agent itself may generate similar and monotonous experiences). By reusing the past transitions, the agent avoids catastrophic forgetting and increases its learning speed. The target network is a neural network with a structure identical to the main model used in the DQN and its purpose is to bring stability to the learning process by replacing the target function (which is frequently changed) by using fixed parameters that are synchronized with the main network only after a defined number of episodes pass. Different reward systems use different reward ranges. While some rewards are scaled in a fixed interval, other rewards can reach very big numbers/ranges. This occurrence makes the learning process unstable and in order to improve this, reward clipping is used. Rewards are usually clipped in a fixed interval which best suits the problem.

Double Deep Q-Network (or DDQN) is an architecture also proposed by DeepMind [8], although Double Q-learning was introduced earlier by the same author [9]. This architecture tries to improve the overestimating of the true rewards that DQN makes. In order to do that, it makes selection and evaluation of actions separately by using the main neural network for determining the best action and the target network for estimating the Q-value. Using the neural networks as two separate Q-value estimators, this architecture provides unbiased estimations of the actions.

The architecture I achieved the best results with is the Dueling Deep Q-Network (or DuelingDQN). It was introduced by DeepMind

[10] as an improvement over the previously described architectures. DuelingDQN explicitly separates the representation of state-values and action advantages by using separate calculation streams. The advantage value of an action represents how advantageous an action is compared to selecting the other actions. The final decision is made based on both state-value and action advantage.

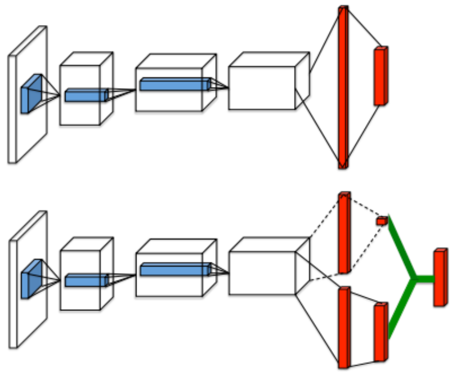


Figure 2: Visual representation of DQN (up) and DuelingDQN (down). The upper stream of the DuelingDQN is used for calculating the state-value and the other for calculating the advantage of actions.

D. Experiments

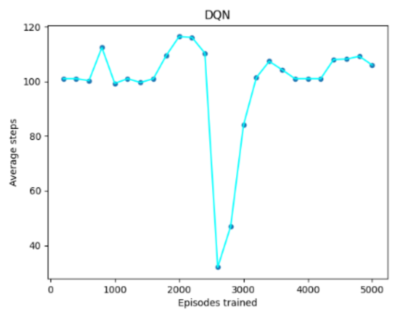
My first step was trying out the above-mentioned architectures and compare the first results. In this step I used the in-built reward system and the RGB version of the environment. Since the state is yielded as an image, I used a model with a few convolutional layers in order to perform feature extraction. Since I was using my local machine for training (I also tried using Google Colaboratory, but it goes to sleep after a certain amount of time), the process took very long. After a moderate number of episodes (which were not enough) and a few parameter modifications, no notable improvements were made. Since I couldn't experiment with a bigger number of episodes, I decided to try the second version of the environment which yields a vector as the state instead of an RGB array.

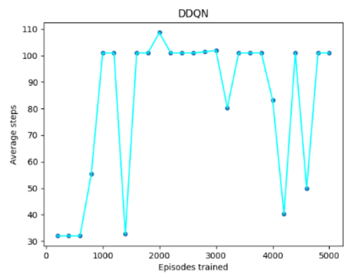
Next, I used the in-built reward system with the second version of the environment using the three architectures. I tried out different parameters and structures for the used neural network (for example a two hidden layer neural network worked better than a network with one hidden layer). The training again took a long time but was more tolerable than previously. After testing the models and plotting the results I noticed that the DuelingDQN achieved the best results. Since the learning curves of the models didn't seem smooth (as if there were some kind of oscillations), I tried training the models with a smaller learning rate, but the agents just stopped learning that way.

I continued experimenting with DuelingDQN since it produced the best results and tried combining it with my custom reward systems. This way it produced much better results. In order to make a better conclusion, I used more shallow and more deep models, different sizes of the replay memory and the batch size and both smaller and bigger learning rates and discount factors. The results of these experiments are presented in the next section.

IV. RESULTS

After training the agents using the second version of the environment and the in-built reward system I plotted the average number of steps that the agent made in 100 tests. Every 200th model was tested (nth model refers to the model saved in the nth episode). I measured the success of an agent in taken steps instead of total reward, so agents trained on different reward systems could easily be compared.





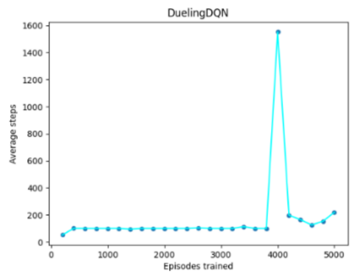


Figure 3: The performances of DQN, DDQN and DuelingDQN with the in-built reward system over 5000 episodes

As it can be seen in Figure 3, DuelingDQN outperforms the other two architectures. It can also be seen that the learning curves seem very unnatural, which is due to the randomly generated environment and random sampling of experiences. Based on this first run on training I decided to continue experimenting with DuelingDQN and the other reward systems. First, I tried the reward system which punishes the agent based on its height difference from the next gap.

reward = 1 – abs(height(agent) – height(gap)) (1)

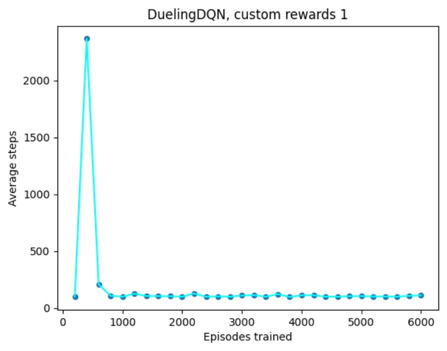


Figure 4: The performance of DuelingDQN using a custom reward system (1)

This agent outperforms all of the previously trained agents. Using this reward system, the agent requires much less episodes to reach its maximum score, so I made another experiment using different parameters and plotted every 50th model instead of every 200th, in order to explore the learning curve. Although this new agent didn't perform nearly as good as the previous, we can see that this reward system encourages a smoother learning process than the in-built reward system.

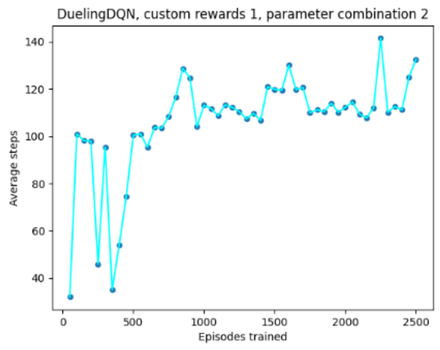


Figure 5: The performance of an agent with the same structure as in Figure 4, but with a different parameter combination

Since I believed that this reward system encourages the agent to learn more smoothly, I tried making another version of the reward system where the negative value of the difference of the heights of the agent and the next gap is multiplied by 10 before calculating the reward.

*reward = 1 – abs(10 * (height(agent) – height(gap)))* (2)

This way, the effect of the height difference penalty is amplified. This agent achieved even better results than the previously trained agents and it managed to reach a point in the learning process where it can reach as much as 4875 average steps.

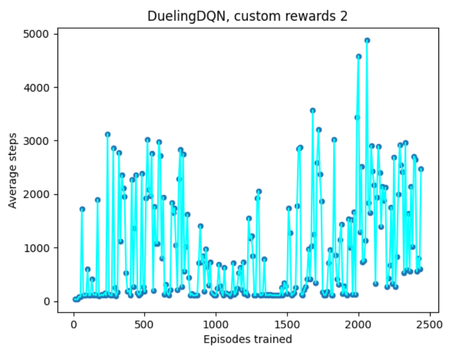


Figure 6: The performance of a DuelingDQN agent using reward system (2)

After this, I decided to try another custom reward system which works the same as (2) but introduces a terminating punishment of -10.

*reward = 1 – abs(10 * (height(agent) – height(gap))) if not done else -10* (3)

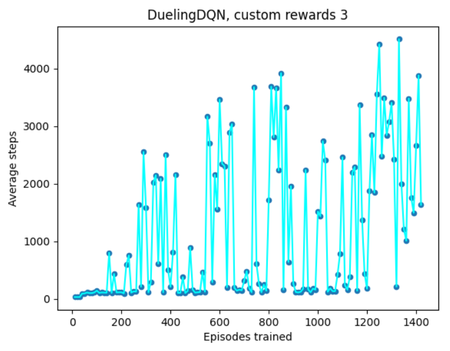


Figure 7: The performance of a DuelingDQN agent using reward system (3)

This training produced good results, comparable with the previous attempt. The agent reaches over 3000 and even 4000 average steps in multiple points of the training process. Also, with using this reward system, the learning curve becomes a bit smoother. Since this reward system proved to work good in this environment, I tried to use it again with the DQN to see how it will perform, but no notable results were achieved.

In order to compare the performances of the tested models, I extracted the best model trained on every reward system. The results can be seen below in Figure 8.

Reward system	**Architecture**	**Average steps**
In-built	DuelingDQN	1551.47
Custom 1	DuelingDQN	2366.91
Custom 2	DuelingDQN	4875.12
Custom 3	DuelingDQN	4510.16

Figure 8: Comparison of the performances of the best models

V. CONCLUSION

After experimenting and testing various models it can be concluded that the in-built reward system can lead the agent to underperform when using the simpler version of the environment. Custom reward systems proved more suitable, especially the ones that acknowledge the height difference between the agent and the next passing point. Although DuelingDQN produced the best results in this project, the other architectures may prove as efficient in case of using images as state representations instead of the used vectors.

I believe the agent can be trained not to lose at all (survive for an infinite number of steps) by using the whole RGB array as a representation of the state (with a suitable preprocessing applied). In that way, the agent can learn to plan its next moves earlier than it does in my experiments. In the version which I use, the agent is able to see information only about the next obstacle, and only when it passes that obstacle it gets information about the next obstacle, which does not give the agent time to plan its steps beforehand and adjust on time (leading to hitting the obstacle). If the agent was able to see the next two obstacles (as it can be seen in Figure 1 - right) it can plan its actions beforehand and significantly improve the overall performance (even with more simple architectures such as DQN).

REFERENCES

[1] Flappy Bird, Wikipedia (https://en.wikipedia.org/wiki/Flappy_Bird)

[2] FlapPyBird, Github (https://github.com/sourabhv/FlapPyBird)

[3] Reinforcement Learning in Python with Flappy Bird, Anthony Li (Towards Data Science: https://towardsdatascience.com/reinforcement-learning-in-python-with-flappy-bird-37eb01a4e786, Github: https://github.com/anthonyli358/FlapPyBird-Reinforcement-Learning)

[4] DeepLearningFlappyBird, Github (https://github.com/yenchenlin/DeepLearningFlappyBird)

[5] Flappy Bird for OpenAI Gym (PyPi: https://pypi.org/project/flappy-bird-gym/, Github: https://github.com/Talendar/flappy-bird-gym)

[6] Deep Reinforcement Learning for Flappy Bird, Kevin Chen, Stanford (https://cs229.stanford.edu/proj2015/362_report.pdf)

[7] Playing Atari with Deep Reinforcement Learning, Volodymyr Mnih et al. DeepMind (https://arxiv.org/pdf/1312.5602.pdf)

[8] Deep Reinforcement Learning with Double Q-learning, Hado van Hasselt et al. DeepMind (https://arxiv.org/pdf/1509.06461.pdf)

[9] Double Q-learning, Hado van Hasselt, Centrum Wiskude & Informatica (https://proceedings.neurips.cc/paper/2010/file/091d584fced301b442654dd8c23b3fc9-Paper.pdf)

[10] Dueling Network Architectures for Deep Reinforcement Learning, Ziyu Wang et al. DeepMind (https://arxiv.org/pdf/1511.06581.pdf)