Machine Learning Engineer Nanodegree

Capstone Project

Natalia Tishaninova
August 13, 2019

I. Definition

Project Overview

The Capstone Project is dedicated to solving a Reinforcement Learning problem.
The main goal of the project is to teach the agent to play Flappy Bird game. The agent at first does not know what this game is about and what should it do to succeed. During training, the agent takes different actions and receives rewards in return. After playing many games, it learns about the environment from its own experience and can decide which action it needs to take at some point of the game to maximize the cumulative reward. It's kind of similar to how do people learn and discover the world around us, which is pretty exciting!

Problem Statement

Flappy Bird is a side-scrolling game where the agent must successfully navigate through gaps between pipes. The agent could take one of two possible actions: 0 - do nothing, 1 - go up.
For each pipe it passes through it gains a positive reward of +1. Each time a terminal state is reached, it receives a negative reward of -5. The game ends if the player makes contact with the ground, pipes or goes above the top of the screen. So the goal is to keep playing as long as possible, avoiding obstacles.
In this project, I would try to solve this problem using Reinforcement Learning algorithms, such as Q-Learning and Double Deep Q-Learning network.

Metrics

To perform the final evaluation, we'll run 100 episodes using a trained model with no random factor in the decision-making process. The best metric is the model's score. To prevent possible infinite episode, we'll force stop episode running if model reaches 300 points (it's tough to achieve this score, considering the highest score on the OpenAI Leaderboard is ~260 points, so probably we don't need to worry too much about infinite episodes). We'll save the best score and the mean score. Visualization of all scores and the final performance will also be a good idea.

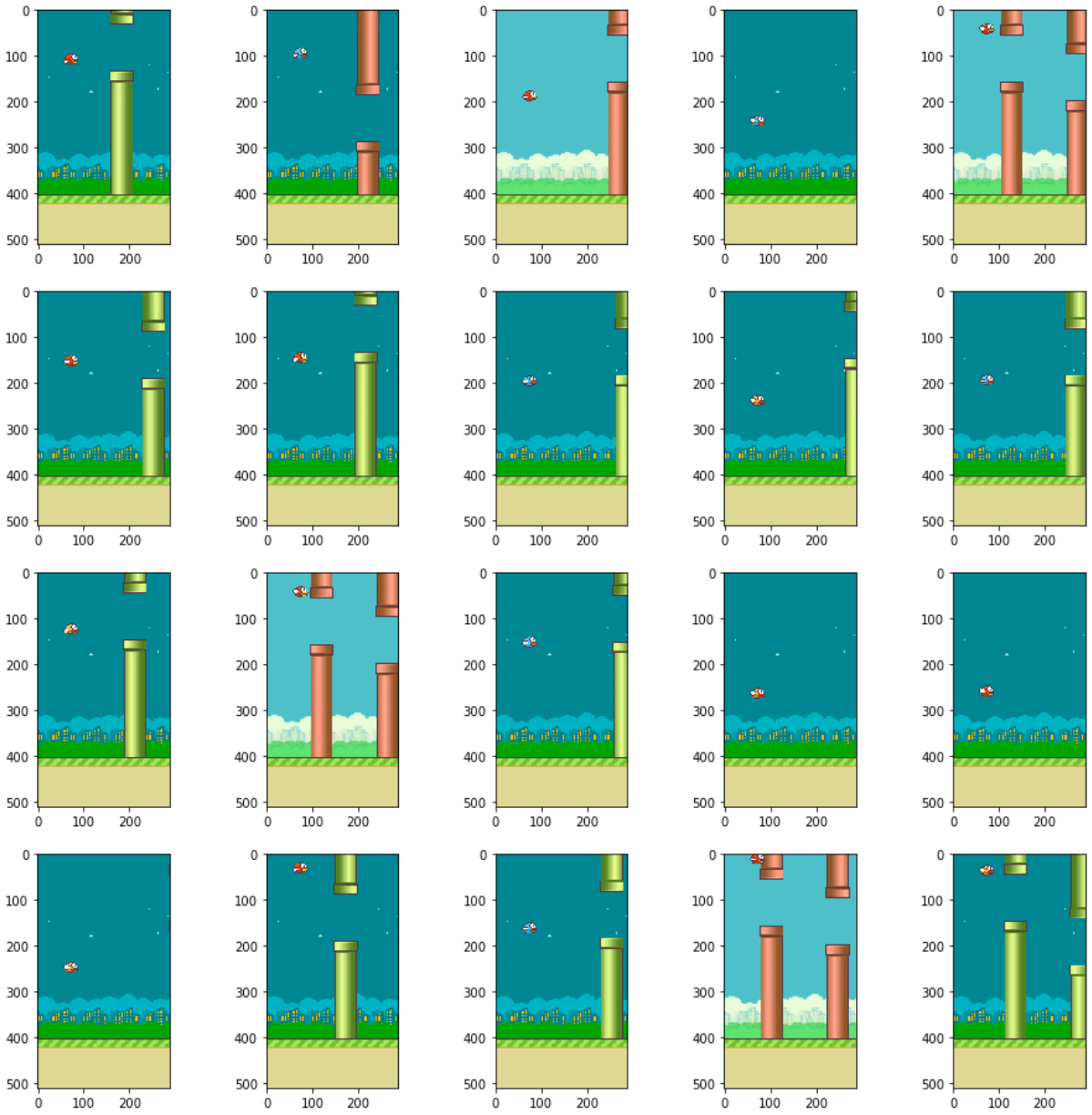
II. Analysis

Data exploration and Exploratory Visualization

In the case of Reinforcement learning data exploration is mostly an environment exploration. We can get information about what's going on in two ways:

1. Get RGB screenshot of size 288x512 of the current game state by calling `getScreenRGB()` on a PLE instance.

You can see some examples of the described function output below:



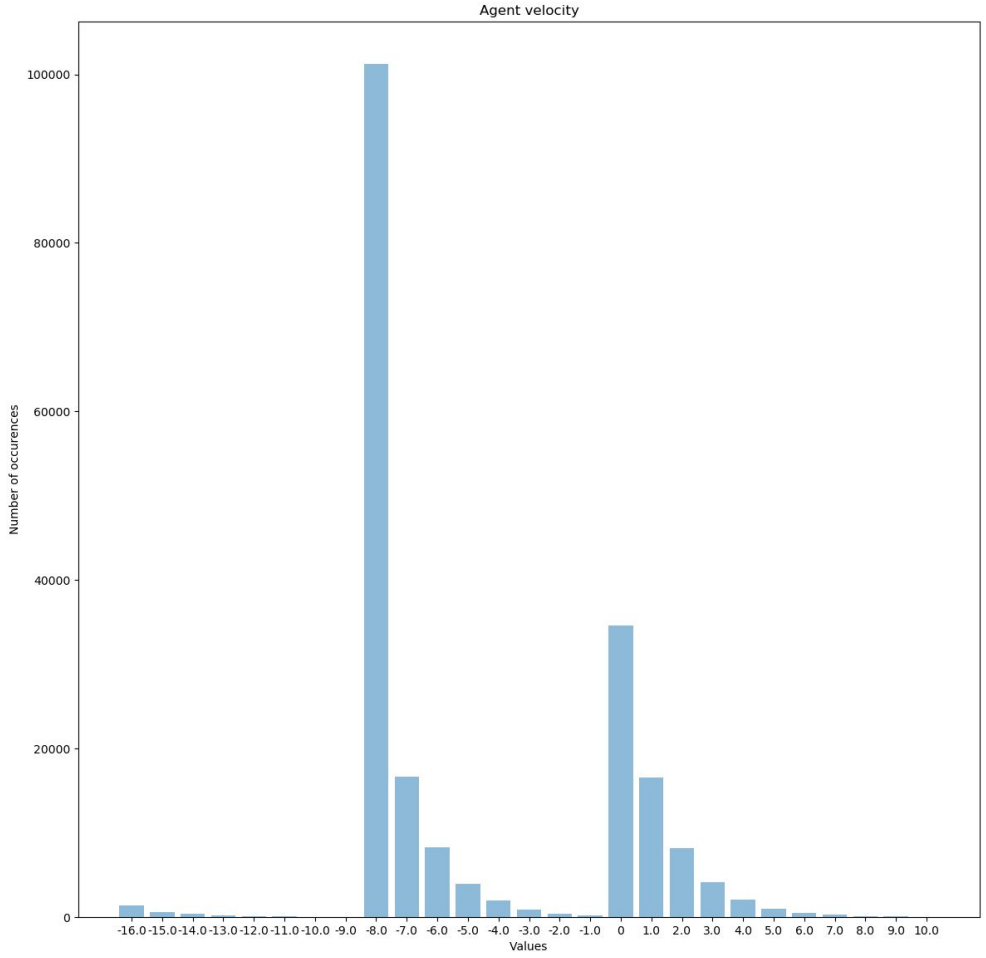
2. Get game state metadata, by calling `getGameState()` function. The output of the function:

- player y position
- player velocity
- next pipe distance to player
- next pipe top y position
- next pipe bottom y position
- next next pipe distance to player
- next next pipe top y position
- next next pipe bottom y position

Let's discuss each of these states in more detail.

Player y position is a position of the bird on the y-axis at the current timestamp (its x-position is constant). This parameter has int value which varies from -16 (the bird collapsed with the screen top) to 350 (the bird fell down and collapsed with the screen bottom).

Player velocity is an integer value between -16 and ten which is negative if the agent moves up and positive if it moves down. Zero value means that the agent moves horizontally. The distribution of player velocity values collected on 200000 frames of random play is shown below.



Negative velocity is more frequent in random games because agent performs ‘up’ action too often, which results in collapsing with the screen top. The velocity of 10 is just the game’s gravity, and it means that the agent is falling.

Next pipe distance to player, as well as *next next pipe distance to a player*, are integer values, which have ranges of [1; 309] and [145; 453] respectively. Their meaning is pretty apparent: x-distance from the agent to the pipe.

Next pipe bottom y position, next next pipe bottom y position (both in range of [125, 292]), *next pipe top y position and next next pipe top y position* (both in range of [25, 192]) values indicate the y borders of the pipe gap.

Algorithms and Techniques

Initially, I was inspired by this paper, so I’ve decided to use Double Deep Q-network to solve this problem.
However, instead of raw game frames, I was using game metadata to train the network.
My network has the following architecture:

Input layer	Game state of shape (4)
Hidden layer	200 nodes
Activation	tanh
Hidden layer	200 nodes
Activation	tanh
Output layer	shape (2)

As it’s Double Deep Q-network, we also create the target network with the same structure and copy parameters from the main network to the target network once in 50 frames to avoid the overestimation problem. For more details, see https://arxiv.org/abs/1509.06461

As we have only two actions, we have only two output nodes. Each represents the value of the given action at the given state. During training, we want to minimize TD error. To do this, I’ve used Adam optimizer with learning rate 0.00001, batch size of 1024, and discount factor (gamma) of 0.99.

Benchmark

As a benchmark model I’ve decided to choose the most straightforward solution to this problem - Q-learning with discretized observation space.
The original continuous state space makes it extremely hard to train a model using simple Q-learning. So we convert the continuous state elements into their discrete representations.
We use the following amount of bins for each state element:

player y - 10 bins
player velocity - 10 bins
distance to the next pipe - 20 bins
next pipe center y - 10 bins

We'll divide each state at equal parts to make things easier and also because we can't tell that there are some obvious outliers in state space (even if some states occur more often during collecting the sample data, let's not forget that we've got this data from a random play, which is chaotic and not fully representative).

Even if this model performs poorly, it demonstrates how complex the problem is, and that simple algorithm can't reach the high-level performance.

III. Methodology

Data pre-processing

After calling `PLE.getGameState()` function we receive a dictionary, containing all available for us information in the following form:

```
{
“player_y ”: float
“player_vel”: int
“next_pipe_dist_to_player”: float
“next_pipe_top_y”: int
“next_pipe_bottom_y”: int
“next_pipe_dist_to_player”: float
“next_pipe_top_y”: int
“next_pipe_bottom_y”: int
}
```

To reduce dimensionality and convert this dictionary to a tuple, let's extract player y, velocity, distance to the next pipe, and also calculate the next pipe center on the y axis using formula: `next_pipe_center_y = next_pipe_top_y + (next_pipe_bottom_y - next_pipe_top_y) / 2`

Implementation

Pipeline

`train.py` - the script that should be executed to train a model.
Parameters:

- --batch_size, -b - size of a minibatch, default is 1024
- --gamma, -g - discount factor, default is 0.99
- --model, -m - Model type to train.

`environment.py` - game wrapper to easily change learning environment implementation if needed. Also contains all code related to reward state and reward processing.

`q_learning.py` - contains an implementation of the discretized version of simple Q-Learning algorithm

`dqn_simple.py` - contains implementation DDQN

`utils.py` - helper functions

`epsilon.py` - everything related to Epsilon-greedy policy

`replay_buffer.py` - contains `SimpleBuffer` for regular state storing and `ReplayBuffer`, which is an optimized version for storing image state

`evaluate.py` - script for model evaluation.
Parameters:

- --model, -m - Model type to evaluate
- --model_path, -path - path to a saved model
- --gif_path - path to a folder, where to store the gif output
- --num_episodes, -n - number of evaluation episodes (default = 100)

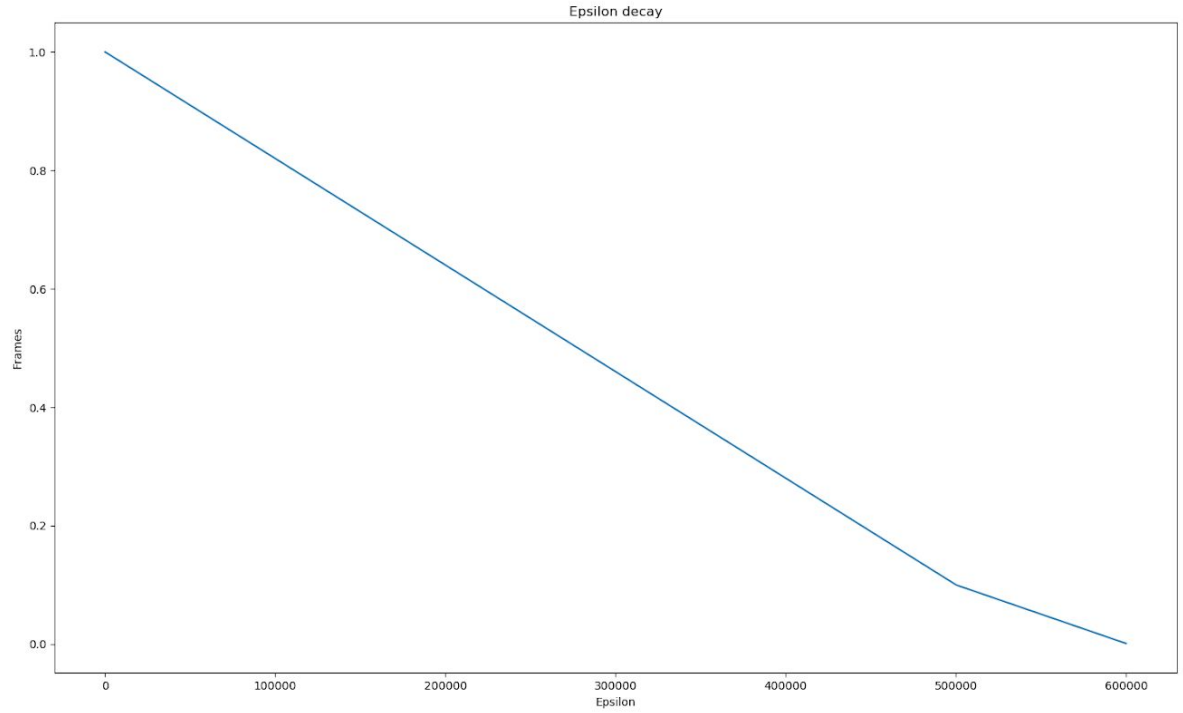
Scripts for training using image input:

`image_transformer.py` - helper class for processing RGB input.

`ddqn.py` - Dueling Double Deep Q-Learning implementation.

Exploration-exploitation tradeoff

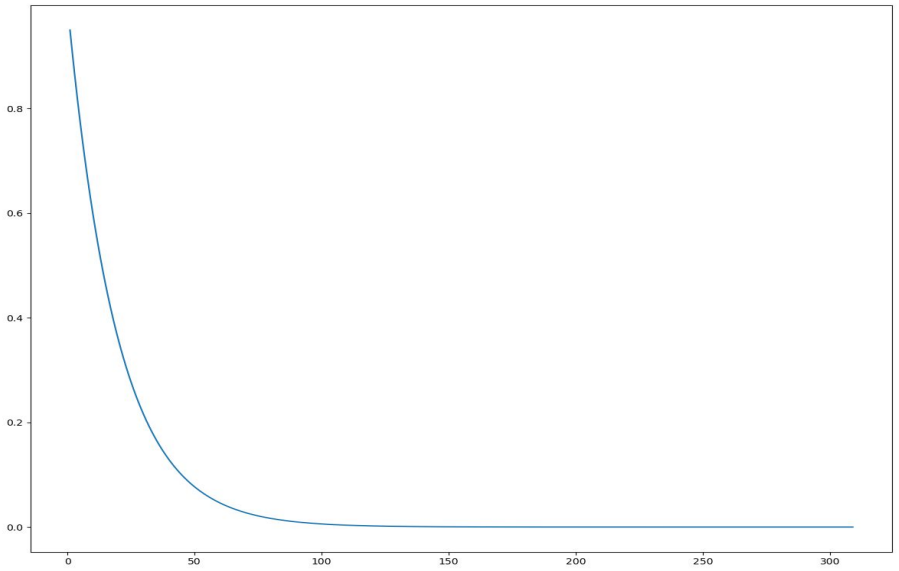
To explore all possible actions in all states and develop an optimal policy, we need to maintain an element of randomness during the decision making about the agent's next action. The epsilon-greedy policy helps us with that: with probability epsilon each iteration, we select a random action. Otherwise, we chose the action with the highest value.
For more efficient training, it's better to start with high epsilon value and slowly decrease it to some minimal value.
Class `EpsilonGreedyScheduler` is responsible for the calculation of epsilon at a particular timestamp of the training. It needs `max_frames` and `epsilon_annealing_frames` as parameters.
For training which lasts `max_frames` we're going to decrease epsilon from 1.0 to 0.1 during `epsilon_annealing_frames` and then reduce it to the minimal value of 0.001 all remaining episodes.
Check out epsilon decrease with `max_frames=600,000` and `epsilon_annealing_frames=500,000` at the plot below.



Reward

The main challenge in training FlappyBird agent is it's the originally sparse reward. It's extremely hard for an agent to figure out what does it need to do if it receives a reward only after successfully passing a pipe. Considering a pretty narrow pipe hole and randomness in the decision-making process at first, it could we a while until agent generalizes all of this and finds a good strategy. This sparseness also leads to very unstable training, which has all the chances never to succeed.

To fix this, we'll use our main advantage - knowledge about the environment. We just need to encourage an agent to move in the right direction. So we'll calculate an 'ideal' point where we want our agent to be (in the center of a pipe hole, slightly shifted to the right). At each timestamp, we'll calculate the distance between the agent and this point and gradually increase the reward for approaching it and decrease reward otherwise, as it's shown at the plot below.



I've also increased the penalty for death to -300 (for training only) and increased reward for the successful pipe passing to 5. This helped to stabilize learning and converge faster.

Refinement

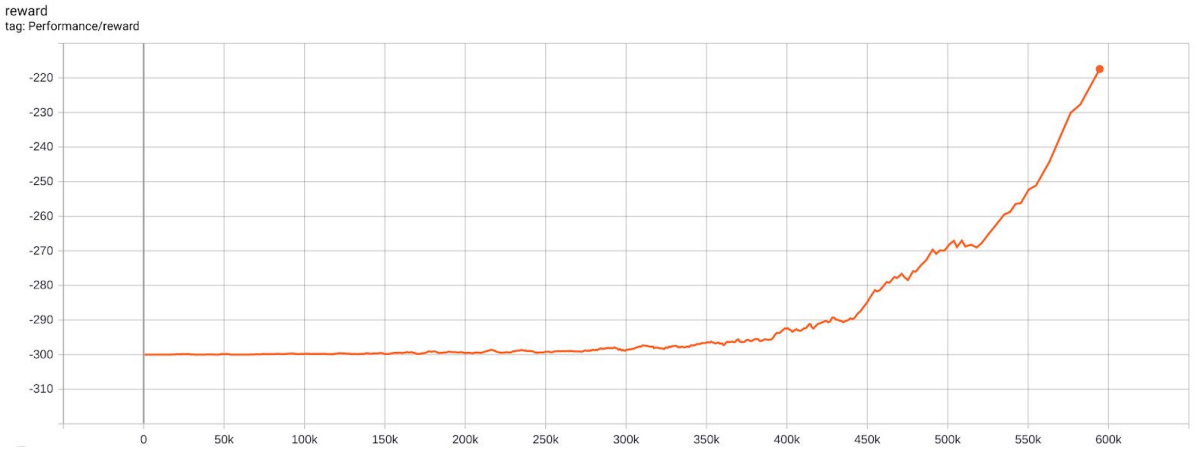
The initial approach was to build a DQN, which would take only the game screen image as input. However, this approach wasn't successful for me: the agent wasn't learning at all.
After the initial failure, I've switched to using the game metadata as an input. It simplified the learning and helped to fasten it (in theory). But the results were still disappointing (the agent wasn't able to get through even two pipes in a row on its own), so I've decided to change the reward. The experience from the Quadcopter project helped a lot. After overwriting the reward function, the agent started to show real progress! So now I've just needed to tune the hyperparameters, be patient, and wait until the end of the training.

Benchmark model choice was also a problem: it should be either naive and straightforward (but not as simple as random play) or piece-of-the art. As I wasn't able to build and train the piece-of-the art model during such a short time, I've decided to go with a simple solution.

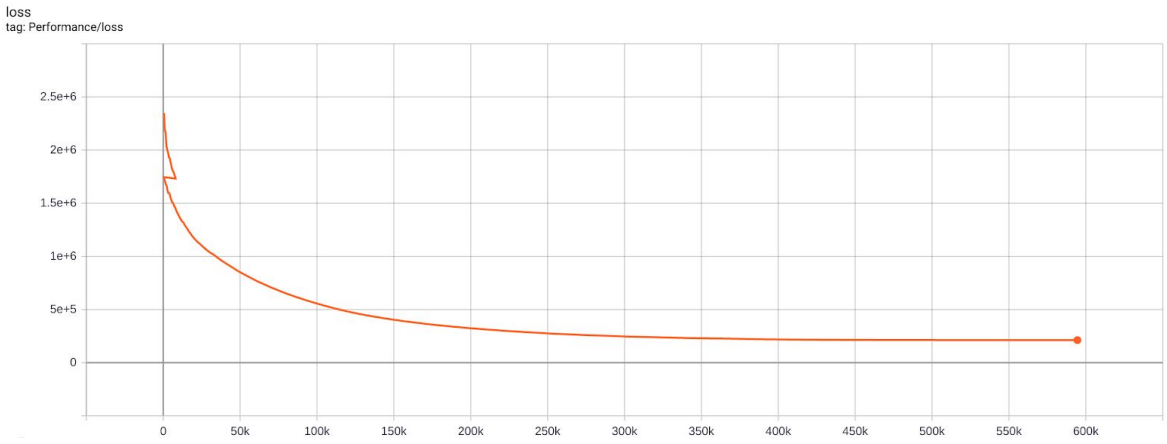
IV. Results

Model Evaluation and Validation

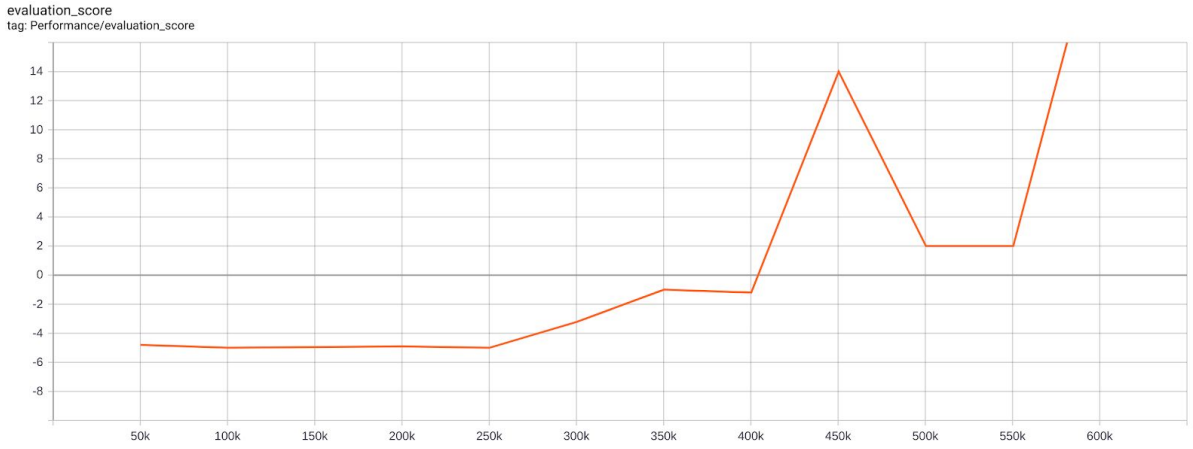
The model was trained for 600,000 frames, which is ~6000 episodes. The graph below shows episode rewards during training.



As we can see, the episodes rewards plot shows constant growth, which means that the agent learns something. The loss is decreasing too, which is displayed on the graph below.



Every 50,000 frames we perform validation gameplay with epsilon = 0 fo 1000 frames to evaluate how the model performs ‘on its own’ without the randomness in actions. Also, during validation, the original rewards were used.
The results are shown on the graph.



Even though the evaluation results are a bit unstable, but they are increasing over time.
After the training was finished, I've tested the model by 100 episodes of gameplay with no randomness and with the original rewards.
The maximum score reached was 219.0, which means that the agent was able to get through 224 pipes before the crash.
Check out the visualized game with the maximum score here.

Justification

Let's compare the performance of our model with the benchmark model (after 100 episodes of evaluation gameplay):

	DQN model	Benchmark model
Mean score	47.18	-4.85
Maximum score	219.0	-4.0

As we can see, DQN agent has much better performance, and it demonstrates that the agent has actually learned to play the game. From the video, we can see that it carefully flies around all obstacles and does everything to reach the next pipe center, which is exactly the behavior I wanted it to have at the beginning of the training.

V. Conclusion

As the goal of this project was to teach the agent to play the FlappyBird game, as we can see from the evaluation video, mission accomplished!
The main points from this project:

- This problem is much more complicated than it seems. I was planning to solve it using only pixel input easily, but this approach did not work for me.
- The problem could be solved with only part of the game state data, I didn't use the information about the next next pipe, and the agent was still able to make good decisions.
- The sparse reward could significantly slow or even make the learning impossible. So it's always a good idea to turn in into gradual reward, which "points" the agents in the right direction.
- It takes an incredibly long time to finish the training, and we can't understand right away whether the agent is learning something or is it just a random success. Also, it makes hyperparameters tuning more difficult.

Improvement

There are also a lot of improvements to this model, which could make the performance even better:

- We can run training “as is” for a few thousands of episodes, it would probably still make some progress.
- we can deepen the network, which could help to improve the action value approximation
- Dueling Q-network or Prioritized experience replay could also help.
- Handle the case when two actions have the equal value: instead of using `max` function, choose randomly between these actions. This would make model more stochastic.
- If we want to play around with pixel input, we can divide this task into two parts:

1. Create a CNN, which would have the pixel image as input and predict the game state (agent y position, velocity, etc.)
2. Combine the above model with an existing model, which can play using game metadata.

So there is a long way to go; the model is far from perfect and needs improvement. It just means that this is a challenging task, and I’ve enjoyed it a lot.