Implementation and Training of a Q-Learning Neural Network for Controlling a Flappy Bird Agent

Cioată Ioana-Larisa Vlad Adriana

January 2025

1 Introduction

This report describes how we created and trained an agent to play Flappy Bird using a neural network and the Q-learning algorithm. Q-learning, a reinforcement learning technique, enabled our agent to learn optimal policies by estimating the expected future rewards for each action in a given state. In our implementation, the neural network approximates Q-values, replacing the traditional Q-table to effectively handle large state spaces.

The solution uses a Deep Q-Network (DQN), a neural network architecture designed for reinforcement learning tasks involving complex state representations. The agent relies on a Convolutional Neural Network (CNN) to process pixel-based input directly from the game environment. The CNN extracts key features from the input, enabling the agent to approximate Q-values for each possible action and make informed decisions. To increase the stability and efficiency of learning, experience replay has been integrated into the training process, allowing the agent to store and reuse episode transitions for off-policy learning by randomly sampling from a replay buffer.

This report details the implementation and results of our approach. We begin by presenting the neural network architecture and its integration with the Q-learning algorithm. Next, we detail the preprocessing techniques used to handle input data efficiently and discuss how we optimized hyperparameters and designed the training process. Finally, we present the experimental results, evaluate the agent's performance, and reflect on the strengths and areas for improvement in our solution.

2 Environment Setup

2.1 Environment Used

We used the `flappy-bird-gymnasium` environment available here. The environment provides:

- State representations as numerical observations containing key information about the game state, such as the positions and velocities of the player and pipes, which are used to render the visual game frames.
- Two possible actions for the agent: `0` (no flap) and `1` (flap).

- A reward function to guide the agent's learning. The agent receives +0.1 points for each frame it survives, +1.0 points for successfully passing a pipe, and is penalized with -1.0 for dying or -0.5 for hitting the top of the screen.
- Additional information about the current episode's status, from which we used the done variable, indication whether the game is terminated or not.

2.2 Rendering

The render() function was used to explicitly obtain the environment's frames. The agent interacts with the environment through actions, while preprocessing techniques were used to convert the frames into inputs appropriate for the neural network.

3 Neural Network Architecture

The agent's policy is modeled using a Convolutional Neural Network (CNN). The architecture is designed to process image data directly and output Q-values for the two possible actions.

3.1 Architecture Details

We experimented with different architectures, and we found that adding more Conv2D layers or Fully Connected (=Dense, Linear) layers made it more difficult for the DQN to learn. These versions began converging 200 episodes later than our final version, and performed worse during testing. Other attempts showed that smaller dimensions for the existing layers led to even worse results.

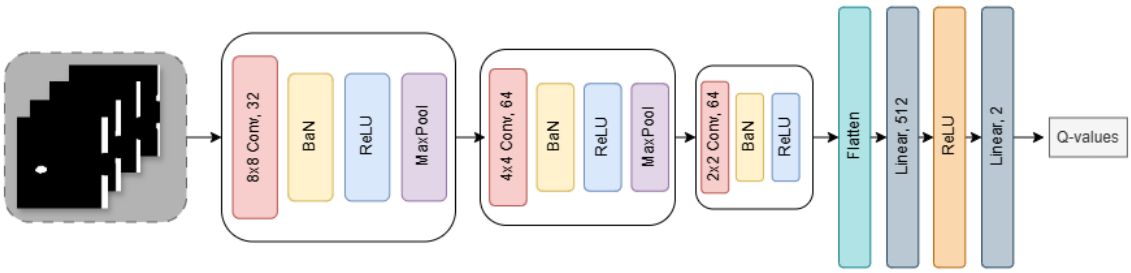


Figure 1: The architecture of the implemented neural network

As shown in Figure 1, the neural network architecture is composed of:

- **Convolutional Layers:**
  - The network begins with three convolutional layers:
    - **First Layer:** Kernel size 8×8, stride 4, padding 2, with 32 filters.
    - **Second Layer:** Kernel size 4×4, stride 2, padding 1, with 64 filters.
    - **Third Layer:** Kernel size 2×2, stride 1, padding 1, with 64 filters.
  - Batch normalization (BatchNorm) and ReLU activation are applied after each layer.
  - Max pooling is applied after the first two convolutional layers.

- **Flattening:**
  - The output of the convolutional layers is flattened to prepare it for the fully connected layers.
- **Fully Connected Layers:**
  - The fully connected section of the network consists of:
    - **Intermediate Layer:** A dense layer with 512 neurons and ReLU activation.
    - **Output Layer:** A dense layer with a number of neurons equal to the number of actions (2 neurons in this case).
- **Weight Initialization:**
  - Weights are initialized using a uniform distribution.
  - Bias values are initialized to 0.01.

4 Preprocessing

Preprocessing of the frames provided by the environment was used to simplify the input and focus on the relevant features of the game. These preprocessing steps, applied in sequence, are as follows:

- **Cropping:** We removed the bottom portion of the frames, specifically what could be seen as the "floor" section, as it did not provide valuable information for training. we found this to be exactly 105 pixels.
- **Grayscale Conversion:** We converted RGB frames to grayscale, as color information does not provide additional value for the agent's decision-making in this task.
- **Resizing:** We resized frames to the desired dimensions for the network. some of the architectures described below used 256×256, others 128×128
- **Background Removal:** The background is removed, by comparing the current frame to the previous one and removing the overlap (using cv2.absdiff) and applying a small threshold of 5
- **Thresholding:** We applied binary thresholding with a value of 128 to further define the objects in the frame.
- **Dilation:** Dilation is used to will in the resulting gaps. we found that a kernel of (5, 5) works best for 128 x 128 and a kernel of (8, 8) works for 256 x 256
- **Erosion:** Erosion is used to define the edges of the objects in the scene. with the same sized kernels, we found the resulting images only "merged" the bird and pipe objects when they were very close. we suspect this was a trait the DQN was able to learn.
- **Normalization:** Scaled pixel values to $[0, 1]$ for numerical stability during training.
- **Frame Stacking:** Four consecutive preprocessed frames were stacked to capture motion, forming an input tensor of shape $(4, 128, 128)$ for the final version of the neural network. This step did not exist for initial architectures and experiments.

5 Q-Learning Implementation

We implemented the Q-learning algorithm with the following enhancements:

5.1 Replay Buffer

A replay buffer stored experiences $(state, action, reward, next_state, done)$ with a capacity of 50,000 transitions in most experiments. Smaller sizes, such as 10,000, led to worse results, while larger numbers like 100,000 caused memory errors. During training, mini-batches of size 64 for most attempts, but 32 for the final version, were randomly selected from the buffer. The stacking of multiple frames per state allowed the DQN to learn well even in smaller batches, something it otherwise struggled to do.

5.2 Target Network

A target network was used to stabilize training by providing fixed Q-value targets. The target network is updated in two scenarios:

- **Periodic Updates:** Every 50 episodes, the target network is synchronized with the main network to reduce oscillations during Q-value updates.
- **Performance-Based Updates:** If the total reward of the current episode exceeds the highest observed reward, the target network is updated to reflect the improved policy.

Updating the target network every slightly more or less episodes didn't result in major changes. Only drastic changes to this number, like updating every 1000 episodes, caused issues.

5.3 Epsilon-Greedy Policy

The epsilon-greedy policy is used to balance exploration and exploitation during training:

- **Exploration:** With probability ϵ, the agent selects a random action. To favor stability, the agent is more likely to select action `0` (no flap) based on a pre-defined ratio (`RAND_ZERO`). The initial value of ϵ is set to 0.5 and decays to a minimum of 0.0001, with a decay factor of 0.9925 applied each episode.
- **Exploitation:** With probability $1 - \epsilon$, the agent selects the action with the highest predicted Q-value for the current state, as estimated by the neural network.

To account for the long-lasting effect of flapping, the epsilon-greedy policy restricts consecutive flapping during the exploitation phase. Specifically, the agent avoids action `1` (flap) if it was performed in the last 8 frames (via a parameter called PATIENCE), based on a comparison between the current frame count and the frame of the last flap. This restriction only applies when the action is determined by the DQN and not randomly.

Yet again, we experimented with many values for these parameters. A starting ϵ of 1 did not assist in exploration, only leading to the network converging later, when ϵ reduced enough, and not any other discernable differences. A decay factor of 0.99 lead to ϵ reducing too quickly, and 0.995 was too slow (ϵ did not reach minimum by the end of training), thus leading us to choose a value between the two.

Values 5, 10 and 8 were tested for PATIENCE. No major differences were observed, though the DQN did perform much better with some PATIENCE than with none. The environment seemed to run at more than 400 frames per second in the right conditions, and seeing how a human should be able to play the game without being expected to click more than 40 times per second, we though a PATIENCE of 10 a reasonable limitation for the network as well.

6 Training

6.1 Hyperparameters

- BUFFER_SIZE = the number of saved past transitions, used for training the model. final version used 50.000. When the buffer is full, new transitions replace the oldest ones.
- BATCH_SIZE = the number of samples from the buffer used for the minibatch in training, once per frame. Final version used 32
- GAMMA = 0.95 across all attempts, used during the Q learning algorithm, representing the discount factor, determining how much the algorithm prioritizes future rewards over immediate rewards when updating the Q-value for a state-action pair
- EPSILON_START = the initial probability of choosing a random action instead of letting the DQN choose. 0.5 final value.
- EPSILON_END = 0.0001 in most versions, signifying the minimum value ϵ can take during training.
- EPSILON_DECAY = ϵ is multiplied by this value at the end of every episode, final value was 0.9925
- LEARNING_RATE = initially fixed, but later adaptive via a custom scheduler that ensures in goes from its initial value to its end value in a specified number of steps, where a step is taken after every training instance, meaning once per frame. Final version utilizes 1e-4, down to 1e-7 over 2e5 steps
- TARGET_UPDATE_FREQ = number of episodes after which the taget model is updated to be the latest one. final version used 50.
- NUM_EPISODES = after which the training stops automatically, though we implemented a signal handler that allowed us to end training and save the latest model at any time.
- RAND_ZERO = the probability of choosing not to flap when taking a random action, 0.85 final value.
- IMG_SIZE = the resolution of the square image used in training, 128 final value.
- PATIENCE = the number of frames after a flap during which the model isn't allowed to choose to flap again, 8 final.

Other important details about our implementation: we used Adam optimizer and MSELoss. Additionally, during training the latest model is saved at the end of every 100th episode.

6.2 Experimentation Attempts

Several experiments were conducted to improve the agent's performance. These experiments are grouped into two main categories: exploration of neural network architectures and hyperparameter optimization. Figures are included to illustrate the impact of each experiment on performance.

Each plot presented in this subsection includes two lines:

- **Blue Line (Original Rewards):** It represents the raw total reward obtained by the agent for each episode during training.
- **Red Line (Smoothed Rewards):** It represents a moving average of the rewards over a fixed window of 50 episodes.

7 Experimental Results

This section presents the results of several experiments aimed at improving the agent's performance. These experiments are grouped under the following subsections:

7.1 Experimental Attempts

Several experimental attempts were conducted to explore different neural network architectures, preprocessing strategies, and logical constraints for the agent.

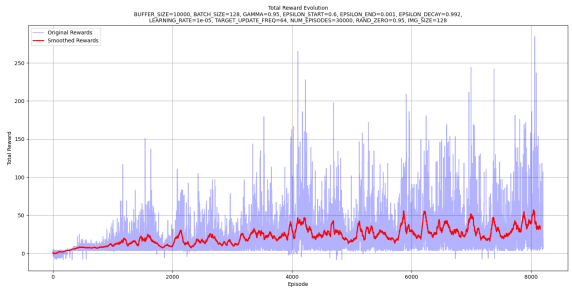
Some attempts are summarized below:

7.1.1 Attempt 1

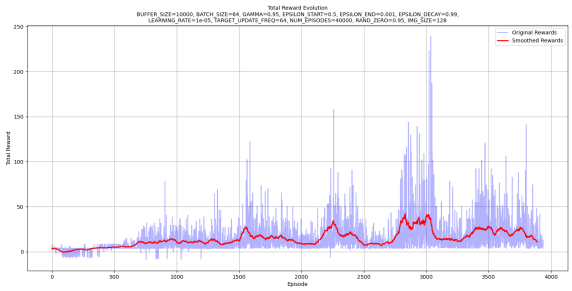
- **Key Features:**
  - Neural network architecture:
    - Three convolutional layers with:
      - Kernel sizes: 8x8, 4x4, and 2x2.
      - Stride values: 4, 2, and 1.
      - Padding: 2, 1, and 1.
      - Batch normalization and ReLU activation after each layer.
      - Max pooling applied after the first two layers.
    - Flattening:
      - The output of the convolutional layers is flattened to prepare it for the fully connected layers.
    - Two fully connected layers:
      - Intermediate layer with 256 neurons and ReLU activation.
      - Output layer with neurons equal to the number of actions.
    - Custom weight initialization with uniform distribution and bias initialization.
  - Other differences:
    - no PATIENCE functionality
    - fixed learning rate

* no frame stacking

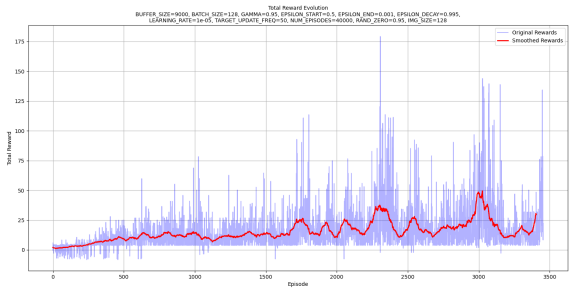
The following figures illustrate the reward evolution for various combinations of hyperparameters tested in this attempt. These experiments aimed to identify configurations that optimize convergence and performance:



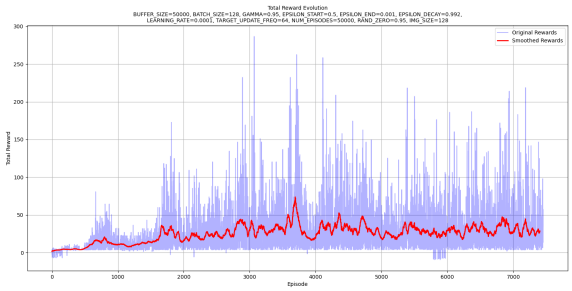
(a) Reward evolution for Experiment 1.



(b) Reward evolution for Experiment 2.



(a) Reward evolution for Experiment 3.



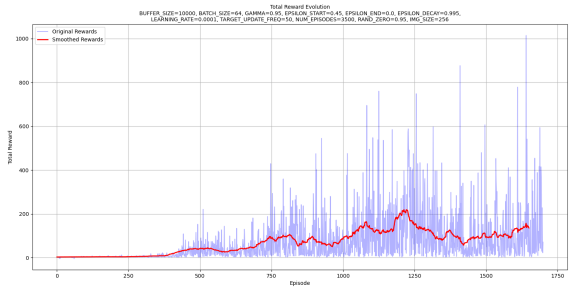
(b) Reward evolution for Experiment 4.

7.1.2 Attempt 2

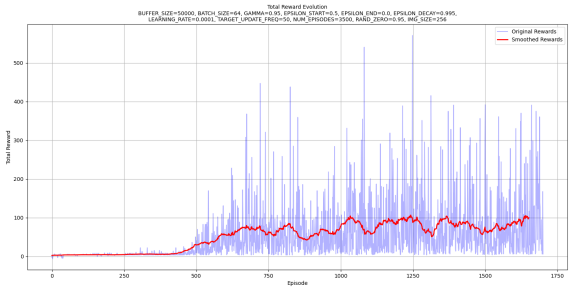
- **Key Features:**
  - Neural network architecture:
    * Four convolutional layers:
      · Kernel sizes: 8x8, 4x4, 2x2, 2x2.
      · Stride values: 4, 2, 1, 1.
      · Padding: 2, 1, 1, 1.
      · Batch normalization and ReLU activation after each layer.
      · Max pooling applied after the first two layers.
    * Flattening:
      · The output of the convolutional layers is flattened to prepare it for the fully connected layers.
    * Two fully connected layers:
      · Intermediate layers with 256 and 32 neurons, both with ReLU activation.
      · Output layer with neurons equal to the number of actions.
    * Custom weight initialization with uniform distribution and bias initialization.

- Other differences:
  - Frames resized to 256×256 pixels.
  - Larger kernels (8×8) used for dilation and erosion operations.
  - implemented PATIENCE functionality
  - implemented adaptive learning rate

The following figures illustrate the reward evolution for various configurations tested in this attempt. These experiments aimed to identify configurations that optimize convergence and performance:



(a) Reward evolution for Experiment 4.



(b) Reward evolution for Experiment 5.

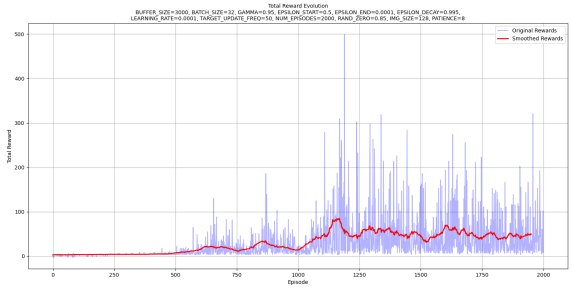
7.1.3 Conclusion from Experimental Attempts

The second attempt achieved higher average rewards within the first few hundred episodes. Improvements to the architecture and the introduction of logical constraints contributed to this progress. However, the results suggest that further improvements are still possible.

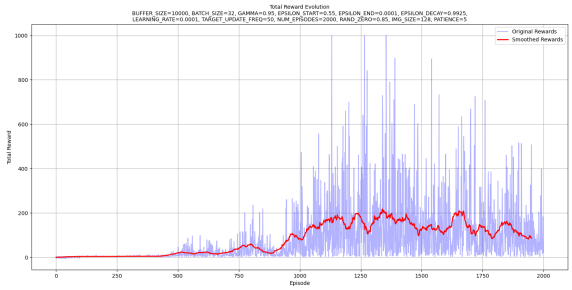
7.1.4 Final Architecture: Hyperparameter Experiments

The final architecture was tested with different hyperparameter adjustments to evaluate their impact on training performance. The variations focused on buffer size, epsilon decay rate, and patience.

The following figure illustrates how some changes influence the reward evolution, to highlight the importance of the values chosen in the end:



(a) Reward evolution for Experiment 6.



(b) Reward evolution for Experiment 7.

These experiments indicate that the tested hyperparameter variations produced different outcomes, generally below those achieved with our configuration. The results highlight the importance of careful parameter selection. Below are the results for the final version:

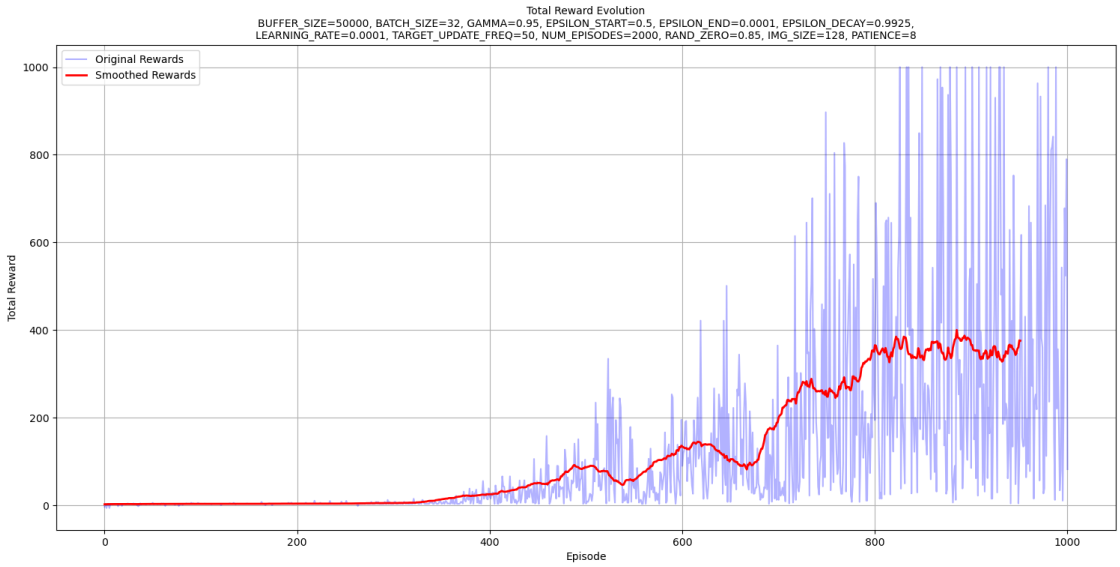


Figure 6: Evolution of rewards during training.

For this version another detail needs mentioning: to get around the theoretical possibility of an infinite episode, we limited the total reward possible per episode to 1000, after which the episode ends even if the game itself doesn't end. Training for this version took around 7 hours.

8 Test Results

So far we've presented how various architectures and parameters behaved during training, where ϵ, although small, could still influence results.

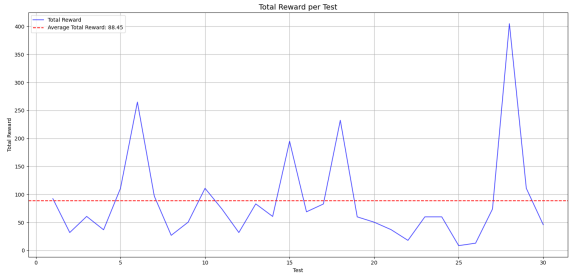
We put aside 7 DQNs and ran 30 tests on each, measuring the reward

8.1 Model 1

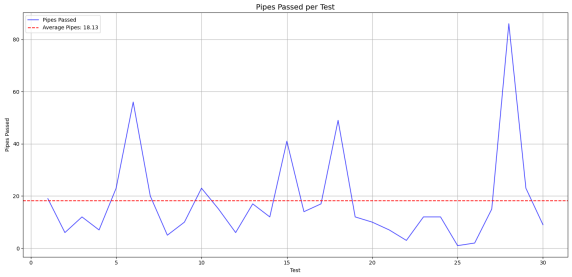
- BUFFER_SIZE = 50000
- BATCH_SIZE = 64
- GAMMA = 0.95
- EPSILON_START = 0.5
- EPSILON_END = 0.0001
- EPSILON_DECAY = 0.995
- LEARNING_RATE = 1e-4, down to 1e-6 over 1e5 steps
- TARGET_UPDATE_FREQ = 50
- NUM_EPISODES = 800
- RAND_ZERO = 0.95
- IMG_SIZE = 256

- PATIENCE 10

This appeared to be the best version before frame stacking, which is why we decided to test it.



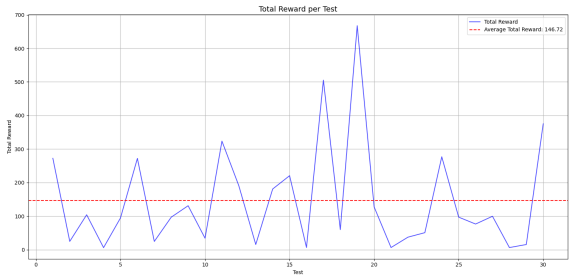
(a) Rewards on 30 tests.



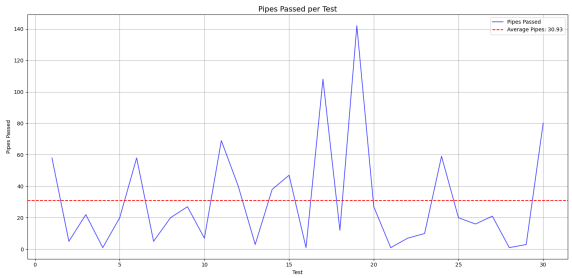
(b) Number of pipes passed on 30 tests.

8.2 Model 2

Similar to the previous, but with a minimum learning rate of 1e-7, over 2e5 steps and trained for 1000 episodes.



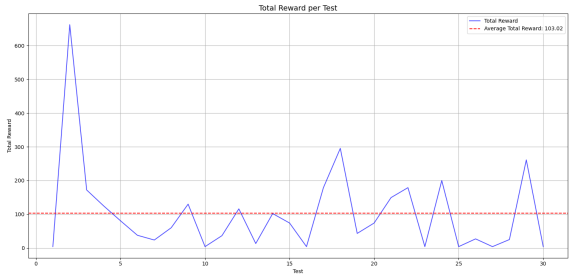
(a) Rewards on 30 tests.



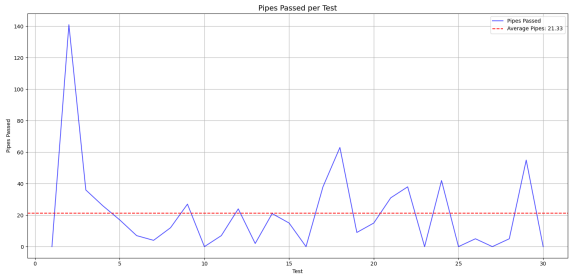
(b) Number of pipes passed on 30 tests.

8.3 Model 3

Similar to the previous, but with a faster epsilon decay of 0.99, trained for 800 episodes.



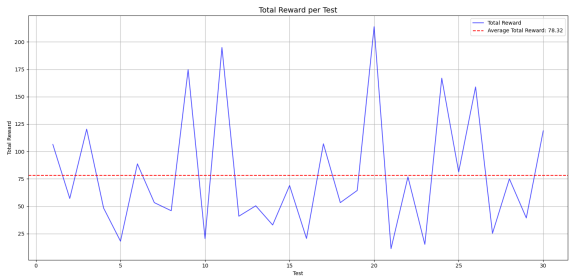
(a) Rewards on 30 tests.



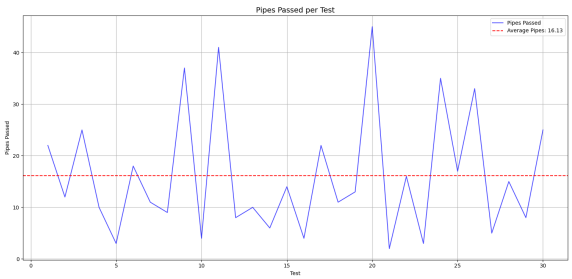
(b) Number of pipes passed on 30 tests.

8.4 Model 4

Similar to the final model, but with bigger 256×256 images, learning rate reduction over 5e5 steps (which proved to be too many, leading to disappointing results), trained for 1000 episodes.



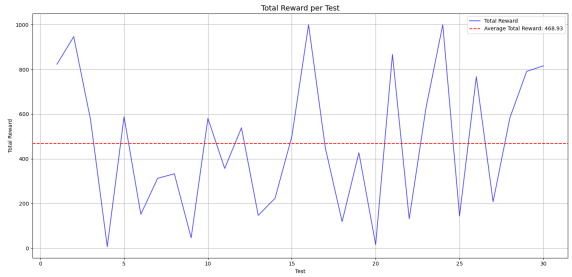
(a) Rewards on 30 tests.



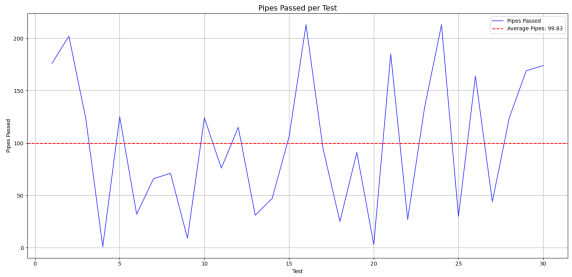
(b) Number of pipes passed on 30 tests.

8.5 Model 5

Final model after 800 episodes



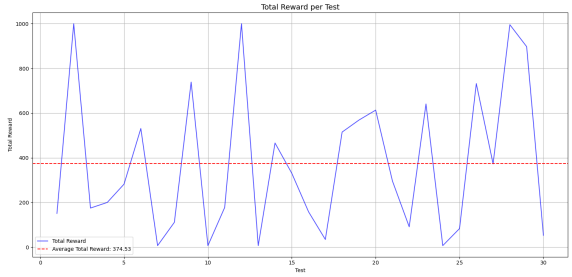
(a) Rewards on 30 tests.



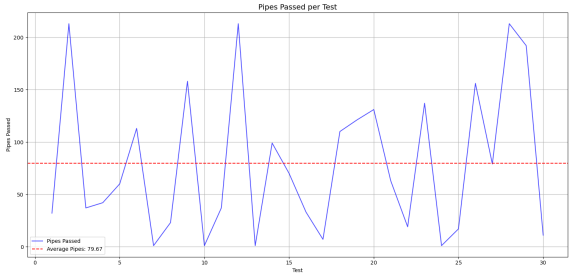
(b) Number of pipes passed on 30 tests.

8.6 Model 6

Final model after 900 episodes



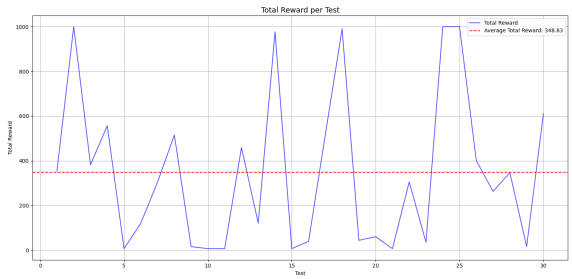
(a) Rewards on 30 tests.



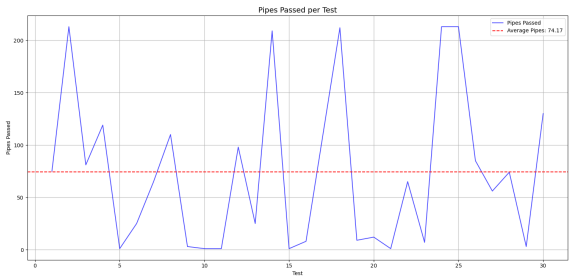
(b) Number of pipes passed on 30 tests.

8.7 Model 7

Final model after 1000 episodes



(a) Rewards on 30 tests.



(b) Number of pipes passed on 30 tests.

As we can see from the final 3 batches of tests, the best average score is not necessarily after more training. In our case most models performed best after around 800 episodes.

9 Conclusion

In conclusion, our work emphasizes the need for careful hyperparameter selection and extensive testing. Small changes, even a slightly smaller minimum learning rate, proved to have a big influence on final results. The most important takeaways from our attempts is that staking frames, while making training much slower and requiring more device memory, allows the model to gain even more information from smaller images than it otherwise could from bigger ones. Also important is that more complex architectures struggled to converge, while simpler ones struggled to learn at all.

We'd also like to mention that we consider our way of frame preprocessing extremely effective, as it was able to very accurately turn the bird and pipes white, and everything else black, extracting and separating the information from each frame perfectly.

Going forward it would be work trying bigger images or bigger buffer sizes than our final attempt, should other devices allow for it, as we believe such changes could greatly increase results. It would also be interesting to try our approach on a different game environment, to see what would change.

10 References

1. `https://sites.google.com/view/rbenchea/neural-networks?authuser=0`
2. `https://edu.info.uaic.ro/computer-vision/`
3. `https://pypi.org/project/flappy-bird-gymnasium/`
4. `https://github.com/yenchenlin/DeepLearningFlappyBird`
5. `https://www.toptal.com/deep-learning/pytorch-reinforcement-learning-tutorial`