Playing Flappy Bird with Deep Reinforcement Learning

Louis-Samuel Pilcer, Antoine Hoorelbeke, Antoine d'Andigne

***Abstract*—We apply to the mobile game FlappyBird a Reinforcement Learning model that learns control policies directly from image observations and from a feedback received when the bird hurts an obstacle. We implement a generic algorithm that looks at the raw pixel values of the game frames and train the agent to take the right decision at any time during the game. We teach a FlappyBird agent how to fly and go through obstacles using a variant of Q-Learning. We investigate the impact of image preprocessing and other ways to improve training time and rewards achieved by the agent.**

I. Introduction

Learning a control policy directly from high-dimensional image inputs in environments that don't provide rewards frequently is a long-standing problem in reinforcement learning [3], that has crucial implications for robotics and autonomous vehicles [5].

Flappy Bird is a famous mobile game in which the player has to guide a bird through the gaps between regularly disposed pipes. The gameplay is very simple : at every instant, the player can choose between two actions : doing nothing, thus letting the bird descend or tapping the screen, thus making the bird fly upward. The general setup of the game can be seen in figure 1.

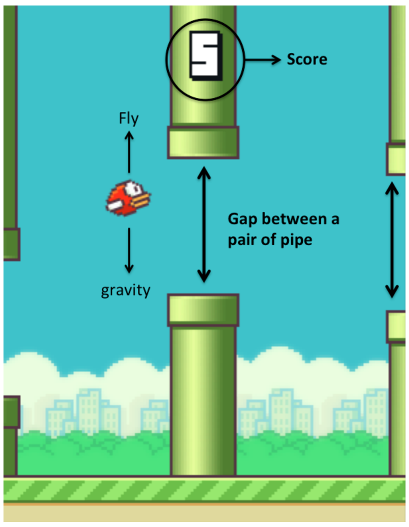

Fig. 1. Flappy Bird setup

II. Background and Related Work

Our project is inspired from Google's DeepMind 2013 paper *Playing Atari with Deep Reinforcement Learning* [6], in which Mnih et al. introduce an agent called Deep Q-Network (DQN). It showed how an AI agent can learn to play a game using screens images without previously knowing anything about the game. They applied the same model to seven Atari 2600 games. The results were pretty impressive considering that in three different games the algorithm performed even better than a human.

We also studied the Stanford ICME paper Playing *FlappyBird with Deep Reinforcement Learning* [7] that applies similar approachs to FlappyBird. They apply a different network structure (2 convolutional layers and 2 linear layers) and test the impact of modifying the loss function. We train our model on 10 times more games, and achieve much better results (we get an average reward of 16.4, and 3.3 for their network). However, their network achieves this average reward playing only 1,600 games. Our method doesn't get any positive reward before almost 8,000 games.

III. The Environment

We used `OpenAI gym` (a game simulator) and the `Pygame` library in order to develop our Flappy Bird environment. A game is a series of actions that translate into observations and rewards. Observations are 512x288 images. The rewards are always 0, and an integer at the end of the game. The integer is based on the number of obstacles Flappy successfully goes through.

IV. Designing our agent

Suppose you are an agent in the Flappy Bird environment. At a given time the environment is in a given state (location and direction of the bird, location of pipes...) that translates into an image such as Fig. 1. At any time the agent can perform two types of actions:

- $a = 0$: do nothing ;
- $a = 1$: fly.

These actions can result in negative reward (the bird crashes before the first obstacle) or in positive rewards (the bird passes some obstacles and crashes between the *reward* and the *reward+1* obstacle) at the end of the game. Positive rewards are based on the number of obstacles the bird passes. When the agent performs any action, the environment changes, leading to a new state.

A. Markov decision process

This set of states, actions and rewards can be represented as a Markov decision process. Each game, can be represented as a finite sequence of states s_i, actions a_i, rewards r_i :

$$s_0, a_0, r_0, s_1, a_1, r_1, ..., s_n, a_n, r_n$$

B. Discounted future reward

At a given state s_i, in order to take into account the future reward and not only the immediate reward, we discount future rewards at a rate γ. Thus the total future reward at a state s_i, if the game stops for *t* = *n*, can be expressed as :

$$R_i = r_i + \gamma * r_{i+1} + \gamma^2 * r_{i+2} + ... + \gamma^{n-i} * r_n \quad (1)$$

Thus, our agent tries to predict, depending on the state, action and policy he adopts, the discounted future reward when it performs an action. In our model we chose to set $\gamma = 0.99$.

C. Q-learning

In order to maximise the discounted future reward at an instant i, we define the action as maximising the expected discounted future reward $Q(s_i, a_i)$ when we perform a_i in state s_i, and continue optimally from that point on. If we assume that we know Q, then it becomes easy to define a policy that decides which action to take :

$$policy(s_i) = \underset{a_i}{\operatorname{argmax}}(Q(s_i, a_i)) \quad (2)$$

The idea of Q-learning is to approximate Q using the Bellman Equation as an iterative update :

$$Q(s_i, a_i) = r_i + \gamma * \max_{a_{i+1}}(Q(s_{i+1}, a_{i+1})) \quad (3)$$

D. Deep Q-network

For the description of a state s_i we used screen pixels that implicitly contain all the needed information (two consecutive screens give the speed and direction of the bird, three give us its acceleration, we thus chose to take 4 consecutive images as inputs).

As the number of possible states is huge (every game shows a different sequence of pipes), we need a way to generalize knowledge we learn on previous games for new ones that will be very different. To approximate Q-values for unseen couples of (*state*,*action*), we build a Deep Q-Network that takes a deep-learning model Q_θ, and tries to find a θ so that Q_θ*(s,a)* approximates *Q(s,a)*.

As our inputs are images, we need a model that is able to learn relevant features on image data and to translate these features into expected Q-values. The best way to represent our Q-function is a convolutional neural network, as these models have an excellent ability to extract relevant features from images [10]. We chose to build a network with 3 convolutional layers that transform our images into more relevant and easier-to-classify features, and then two dense layers with a rectified linear and then a linear function.

Our network takes the state (four consecutive game screens) as input and output an estimation of the Q-value for actions $a = 0$ and $a = 1$. We chose the same network architecture that DeepMind used:

Then we update the weights of the network using the Gradient Descent algorithm on the loss function :

$$loss(i) = \frac{1}{2}[r_i + \max_{a_{i+1}}(Q(s_{i+1}, a_{i+1})) - Q(s_i, a_i)]^2 \quad (4)$$

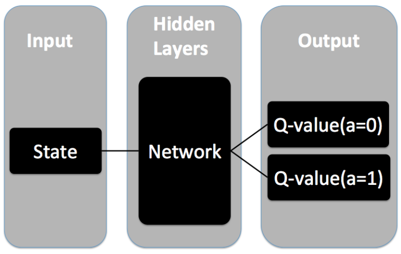

Fig. 2. Deep Q-network illustration

TABLE I
NETWORK ARCHITECTURE

Layer	Nb. filters	Filters size	Nb. units	Activation
Conv. layer 1	32	8*8		ReLU
Conv. layer 2	64	4*4		ReLU
Conv. layer 3	64	3*3		ReLU
Fully connected 1			512	ReLU
Fully connected 2			2	Linear

E. Exploration-Exploitation

The neural network being initialized randomly, it initially acts following a random policy and then will be improved by training until it finds a successful strategy. But the first successful strategy isnt necessarily the best one. The question the network should address is : should I exploit the known working strategy or explore other, that may be better strategies? In fact, the first strategy is greedy as it sticks with the first effective type of policy it discovers.

In order to avoid this problem of sticking with a non-really-effective strategy, we use ϵ-greedy exploration: with probability ϵ choose a random action, otherwise choose the action that maximises the $Q-function$. In our model, just as in Deepmind's model, ϵ value decreases linearly over iterations from 1 (random actions) to 0.1.

This way we ensure to avoid sticking with non-really-effective strategies.

F. Experience replay

In order to stabilize over time the $Q - values$ given by the approximation of the $Q - function$, we used the technique of experience replay introduced by Deepmind [3]. Experience replay consists of storing a defined number of the last experiences

$$s_i, a_i, r_i, s_{i+1}$$

in a replay memory and to randomly use them when running the gradient descent algorithm that trains the neural network. This process of experience replay might reduce oscillations as we train our network not only on recent observations/rewards, but also on data that we randomly sample in the agent's memory. We train our model on batches of data that might represent many past behaviors.

V. Experiments and Results

A. Experiments

We used the deep learning library `Keras` with a `Tensorflow` back-end. We used the open-source Deep Q-Network agent implementation available in `Keras-rl` library to perform fast training.

We launched the training algorithm for 1,500,000 iterations on various configurations. We first tried to train an agent relying on value function approximation methods, and we tried to train a Q-learning agent with a linear approximator, and then with a deep neural network composed of 2 dense layers.

These methods didn't perform at all: after 1,500,000 iterations, the bird still didn't pass any pipe, with an average score close to -1.

We thus implemented, using the Keras-RL framework and Keras/Tensorflow, a Deep Q-Learning agent using a classic computer vision model to approximate Q-values. After 1,500,000 iterations and almost 17,500 games, our model achieved performances that were on average much better than non-expert human players.

As training was very long, we tried to reduce training time by preprocessing images and reducing their dimension. Our assumption was that a preprocessing would facilitate finding relevant features through convolutions.

B. Preprocessing

At first, we trained a Deep Q-Network with basic image features built by gray-scaling our observations.

To make training faster, we built a more complex preprocessing method. We found a way to erase the background image and to keep only the bird, pipes and the ground, with binary features. Our idea was that these preprocessed images would make bird detection easier (just have to find a circular zone where pixels are equal to 1) and that it would enable the model to learn easily the distance between the bird and the pipe, the altitude difference, etc.

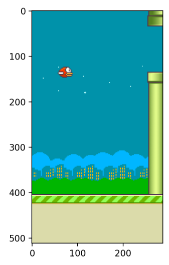

Fig. 3. Image of FlappyBird before preprocessing

C. Results

Our code is available on the Github repository https://github.com/anth2o/inf581_flappy_bird, and a video of our agent playing has been uploaded on Youtube: https://www.youtube.com/watch?v=Tf8SVv1nPxM&feature=youtu.be.

Our code is reproductible and can be adapted to any other environment with image observations by modifying the *gym.make* calls. To run it, you just need Python 3, OpenAI Gym's implementation of Pygame GymPLE (installation instructions: https://github.com/lusob/gym-ple) and to run *pip install -r requirements.txt*. One can test our FlappyBird agent on 10 random games by running the *demo.py* script. Pre-trained models are in *callbacks/*.To re-train an agent, scripts have been written in the *script/* folder.

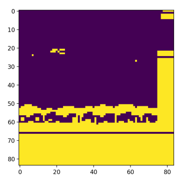

Fig. 4. Preprocessed image of FlappyBird

We compare our models' performances with human scores, referenced in *Playing FlappyBird with Deep Reinforcement Learning* [7].

TABLE II
Results

Environment config.	Human	DQN	Preprocessing + DQN
Average score	4.25	16.4	15.6
Median score		8	6
Maximum score	21	80	65

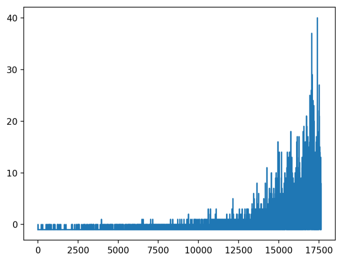

Fig. 5. Average reward during training

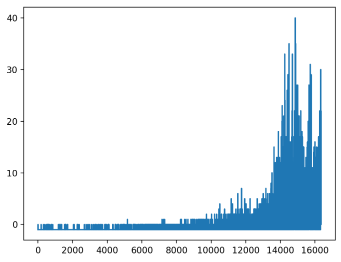

Fig. 6. Average reward during training, with preprocessed features

VI. Analysis and Discussion

A. Analysis

- To learn control policies from high-dimensional image data, a Deep Q-Network agent with a computer vision model can perform much higher scores than humans.
- Training is very long but can be shortened with a well-chosen preprocessing method.
- Rewards start improving very late.
- Preprocessing helps the model learn policies faster: as a reward becoming positive means that the bird successfully passes the first obstacle, the model with preprocessing learns policies with 8,000-9000 games, the model without preprocessing learns with 11,000 games.
- There is a very high variance in the results.
- Variance can be due to the random bird color (red, but also blue or yellow, which could be in some case hardly distinguished from the background). We could preprocess the image in a way that helps the model identify the bird's position to solve this issue.
- It can also be linked with the choice of reward series that could lead to low anticipation. Studying the impact of changing the discount factor on the variance can be interesting.
- Even with experience replay, training is very unstable: new data can lead to decreased scores.

B. Stability

Evaluating a strategy during training is not easy for reinforcement learning, as policy evolves dynamically and we might not have the possibility to run different policies on the same test games in order to evaluate our model during training. Our evaluation metric is thus the average reward on recent games, which is very noisy as small changes in our model's weights can modify actions in a way that leads our agent to visit unvisited types of states where the model does not perform that well. For example, a small change in weights could lead the bird to fly too often and visit high-altitude states where the model does not perform that well as it did not visit a lot of them. This might explain the very high noise noticed on Fig. 5 and Fig. 6.

C. Possible improvements

The first thing we notice when we test the model is the very high variance between games. Our bird can miss the first obstacle for a play, or it can pass more than 80 obstacles. Our assumption is that this variance is due to 2 elements:

- There is a high difference between environments, depending on the initialization. Some cases are different and do not appear very often (high altitude difference between consecutive obstacles for exemple). These cases might need a different kind of policy: if there is a huge altitude difference, the bird might have to fly two or three consecutive times; if the altitude difference is low, it might not need to fly more than once before the next obstacle. This can be solved by training the agent on more game plays, to enable it to learn situation-specific models.
- The bird does not anticipate very well schemes that could lead it to fail passing an obstacle that is still far away. We could test this hypothesis by working on the discount factor, setting it at 0.999 or higher in order to build an agent that anticipates better future rewards.
- We could also work on the reward definition, for example by computing an additional reward every time the bird successfully passes an obstacle. It might imply more work to find the right moment to add this reward, but might give our bird a better understanding of the game structure.
- As a single wrong action can make you lose, keeping a 0.1 epsilon after the exploration phase can be a bad choice. We could test training the agent with a 0.1 to 0.01 linear annealing epsilon.

To make training faster and more accurate, the next steps would be:

- To build a hand-made pipeline that deduces relevant features from the image. For example, preprocessed images make it possible to detect quite easily the altitude of a bird, the space between the bird and the next obstacle and the altitude of the space where the bird can pass the obstacle.
- To find a way to reduce the game observations' dimension. A first idea would be to resize the image, or to resize the image after a preprocessing in order to keep only very relevant feature (in a 40x40 or a 20x20 pixels dimension, for example). The dimension might be well chosen, in order to keep relevant information.
- Another way to reduce dimension before training would be to generate thousands of random games and to build a Principal Component Analysis space on these images, in a lower dimension that keeps most of the variance. We would then train the model on the observations projected on the PCA space.
- We might also work on the structure of the network. There might be a lighter network structure that performs equivalent scores after the same training.

D. Faster policy learning

One possible improvement would be to shorten the ”stagnating phase”: we don't improve rewards during the first 8,000 games.

We make two assumptions about the reasons of this failure to learn policies that make a positive score at the beginning of the game:

- It might be linked with our exploration/exploitation tradeoff. At the beginning, we have a 1-greedy policy, which means that the decision to fly or not to fly is completely random. As a good policy is to fly every 5 to 10 iterations, an efficient way to discover good policies at the beginning of the game could be to modify the *random* decisions, for example to make flying 5 times less frequent.
- It might be linked with the high number of coefficients of our neural network. One idea would be to initialize the game with a network that would be pre-trained, either

for an object detection problem such as *ImageNet* [8], or for a similar game such as *Atari*. We could either retrain the whole model, or keep the convolutions' weights and retrain the dense layers [9].

VII. Conclusion and Future Work

This paper adapts the Deep Q-Network deep reinforcement learning agent to FlappyBird, a game with very little environment feedback and high-dimensional image inputs. We trained two models to study the impact of observation preprocessing on training speed (number of iterations before our agent starts learning efficient policies). One could study this problem further by providing a more detailed analysis of the impact of reward engineering and discount factor. We could also analyze the impact of epsilon's maximum value. Another interesting element that should be studied deeper is preprocessing and dimension reduction, as it might help faster training and features discovery. Finally, we might study the impact of transferring knowledge learnt by a model trained on another similar environment on training speed and performance.

References

[1] D. Barber. Bayesian Reasoning and Machine Learning, *Cambridge University Press*, 2012.

[2] J. Read. Lecture III - Structured Output Prediction and Search. *INF581 Advanced Topics in Artificial Intelligence*, 2018.

[3] Mnih et al., Human-level control through deep reinforcement learning *Nature* 518, pages 529533 (26 February 2015)

[4] O. Vinyals et al. StarCraft II: A New Challenge for Reinforcement Learning. https://arxiv.org/abs/1708.04782, 2017.

[5] Chen et al., DeepDriving: Learning Affordance for Direct Perception in Autonomous Driving http://openaccess.thecvf.com/content_iccv_2015/papers/Chen_DeepDriving_Learning_Affordance_ICCV_2015_paper.pdf

[6] Mnih et al., Playing Atari with Deep Reinforcement Learning https://arxiv.org/pdf/1312.5602v1.pdf

[7] Appiah et al., Playing FlappyBird with Deep Reinforcement Learning http://cs231n.stanford.edu/reports/2016/pdfs/111_Report.pdf

[8] Deng et al., ImageNet: A Large-Scale Hierarchical Image Database http://www.image-net.org/papers/imagenet_cvpr09.pdf

[9] Taylor et al., Transfer Learning for Reinforcement Learning Domains: A Survey http://www.jmlr.org/papers/volume10/taylor09a/taylor09a.pdf

[10] ImageNet Classification with Deep Convolutional Neural Networks http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks