FlappyBiRL: a reinforcement learning agent to play Flappy Bird

Giuseppe Boezio
University of Bologna
giuseppe.boezio@studio.unibo.it

Abstract

Reinforcement learning is one of the most popular approaches for automated game playing. It allows an agent to be able to perform a task based on a trial and error mechanism. In this paper reinforcement learning is applied to the game of Flappy Bird. A2C algorithm was implemented with the standard loss and adding the entropy regularization. Two different notions of state have been compared and it turns out how an approach based on features as vertical and horizontal distance from the next pipe and vertical velocity performs better than an approach based on a stack of images.

1 Introduction

Reinforcement Learning (RL) is an area of machine learning which turns out to be very useful to solve problems where there is no a single way to complete a task. This kind of tasks are very common in the game sector. In fact, in the last decade, several games like arcade ones, Go and Backgammon have been successfully played by AI agents which use RL techniques to defeat their opponents.[Mnih et al. (2013)][Silver et al. (2016)][Tesauro (1994)]

In this paper it is described an agent which can play Flappy Bird mobile game[1]. Flappy Bird is a game for mobile phone became popular in 2014. In this game the player tries to keep the bird alive for as long as possible. The bird automatically falls towards the ground by due to gravity, and if it hits the ground, it dies and the game ends. The bird must also navigate through pipes. The pipes restrict the height of the bird to be within a certain specific range as the bird passes through them. If the bird is too high or too low, it will crash into the pipe and die. Therefore, the player must time flaps/jumps properly to keep the bird alive as it passes through these obstacles. The game score is measured by how many obstacles the bird successfully passes through. Therefore, to get a high score, the player must keep the bird alive for as long as possible as it encounters the pipes.

Different kind of agents have been developed to play this game but most of them use Temporal Difference (TD) algorithms such as Q-learning, SARSA and Deep Q-learning. The purpose of this project is to investigate whether an actor-critic method such as A2C can be used to get successful outcomes for this game and also try to understand which is the best state representation and reward function which can help the agent to learn as fast as possible.

[1]code is publicly available on Github

Preprint. Under review.

2 Related Work

The most used algorithm is Q-learning both tabular and deep learning version; some authors use also SARSA. Another important aspect is the network used to train and evaluate the agent. In this case two main approaches have shown good results. The first one consists of having a vector composed by the horizontal and vertical distance from the next pipe and sometimes the velocity on the y axis to use as input of a multilayer perceptron whereas the second approach consists of adapting this problem to Atari arcade games and using the same architecture and preprocessing steps of [Mnih et al. (2015)]. Following, a table to compare the main projects which have already taken into account this game.

Reference	Algorithm	State	Reward	Network

Chen (2015)	Q-learning	image
		shape = (84,84,4)
			alive = 0.1, dead = -1
			pipe = 1
				CNN

Vu and Tran (2020)	Q-learning
	SARSA
		$(x_{diff}, y_{diff}, y_{vel})$
		$(image, y_{vel})$
			alive = 0.5, dead = -1000
			pipe = 5
				MLP (50-20)
				CNN

Kovtun (2019)	Q-learning	(x_{diff}, y_{diff})	alive = 1
			dead = -1
				MLP (4-4)
				MLP (4-4-2)

Eklavya Sarkar (2016)	SARSA	image
		shape = (84,84,4)
			alive = 0.1, dead = -1
			pipe = 1
				CNN

Lin (2018)	Q-learning	image
		shape = (80,80,4)
			alive = 0.1, dead = -1
			pipe = 1
				CNN

Li (2021)	Q-learning	$(x_{diff}, y_{bottom}, y_{vel}, y_{top})$	alive = 1
			dead = -1000
				tabular

Gong (2020)	Q-learning	(x_{diff}, y_{diff})	alive = 15
			dead = -1000
				tabular

Table 1: Comparison of different solutions

As described in the table 1, $(x_{diff}, y_{diff}, y_{vel})$ denote respectively the horizontal distance from the next pipe, the vertical distance from the center of the next two pipes and the velocity on y axis. y_{bottom} and y_{top} denote the vertical distance from the upcoming bottom pipe and the distance from the top and the upcoming bottom pipe. CNN means Convolutional Neural Network whereas MLP is Multilayer Perceptron. Tabular denotes a non-deep architecture which uses a matrix to store the action-value function.

3 Markov Decision Process

3.1 Model description

As described in the section 1 two different notions of state have been used. They will be dubbed as $state_{vector}$ and $state_{image}$.

$$state_{vector} \in Q^{8\ x\ 3}$$

The three columns represent $(x_{diff}, y_{diff}, y_{vel})$ for eight consecutive timesteps. y_{vel} has not been used during the first experiments because the network which accepts this state as input (further described in section 5.1) uses a recurrent layer and for this reason it was thought that the information provided by the velocity could be replaced by the evolution of the x and y position during the time sequence. However, it turned out how performance without velocity are worse than expected, therefore this feature has been thereafter included.

$$state_{image} \in [0..1]^{84\ x\ 84\ x\ 4}$$

In this case 4 images of shape $(84, 84)$ are stacked together as state to use for the input layer of the neural network. Each image can be obtained through different preprocessing steps described as follows:

$$luminance : \{0..255\}^{288\ x\ 512\ x\ 3} \rightarrow \{0..255\}^{288\ x\ 512}$$

`luminance` is a function which transforms the 3d image corresponding to a screenshot of the environment to a 2d image combining the rgb values to get a luminance measure. The conversion is performed using the following formula described in [MacAdam (1937)].

$$luminance = 0.3 * R + 0.59 * G + 0.11 * B$$

The output of this function is a grayscale image representing the luminous power for each pixel of the previous image.

$$rescale : \{0..255\}^{288\ x\ 512} \rightarrow \{0..255\}^{84\ x\ 84}$$

`rescale` is a function which reduces the size the image to make it compatible with the convolutional neural network (CNN) architecture.

$$normalize : \{0..255\}^{84\ x\ 84} \rightarrow [0..1]^{84\ x\ 84}$$

`normalize` is a function which forces the values of the image to be between 0 and 1.
For each interaction with the environment the image is processed using the compound function:

$$preprocessed\_image = normalize(rescale(luminance(image)))$$

All these steps have been followed as described in the paper concerning Deep Q-learning for Atari games [Mnih et al. (2015)] even if it is not described an explicit formula for the luminance which was computed as previously mentioned.

The actions are $\{flap, nothing\}$ where `flap` = 1 and `nothing` = 0.

The reward function returns `15` for each timestep the agent is alive, `-1000` when it dies as described by [Gong (2020)]. Giving more weight to the death of the agent seems to allow the agent a faster learning with respect to when it is not penalized for the death or penalized with an absolute value equals to the positive case. The pipe reward has not shown better results, therefore it has not been used.

3.2 Infrastructure

The environment has been implemented using the RL library OpenAI Gym [Brockman et al. (2016)]. The implementation realized by [Talendar (2021)] has been used even if it was modified for the following reasons:

- there was not a rigid ceiling in the environment, therefore when the agent was off-screen the algorithm did not work well.
- the reward was set to 1 for each timestep the agent was alive. This caused problems when the agent reached the first pipe because it did not learn by the wrong actions which brought it to die but it could improve only randomly choosing an action to pass through the next pipe. Moreover, to test whether the pipe rewarding were useful or not it was implemented a mechanism to check whether the agent successfully passed the pipe or not.
- when the bird dies, the game restarts with the bird at the same position every time. A random starting position was implemented to allow the bird to explore the state space more efficiently as suggested by [Ebeling-Rump and Hervieux-Moore (2016)].

4 Methods

The choice of an algorithm (A2C) which allows to run multiple agents in parallel relies on the following reason: the execution of multiple agents in parallel turns out to be a good strategy to reduce the drawback of Deep RL algorithms which use experience replay because these ones need more memory and computation per real interaction and update from data generated by an older policy. The parallelism also allows agents to explore a variety of different states at the same time [Mnih et al. (2016)].

The A3C (asynchronous advantage actor-critic) belongs to the class of aforementioned algorithms and it is an actor-critic method. It uses two networks: a policy network (actor) to produce a probability distribution of actions given a state and a value network (critic) to compute the value of a state which will be used as baseline in the algorithm. The two networks usually share most of the layers and the global network produces the two previously mentioned outputs.

A2C (advantage actor-critic) is the synchronous version of A3C which waits for each actor to finish its segment of experience before performing an update, averaging over all of the actors. There is no evidence that the asynchronous version performs better than the synchronous one and in some cases the implementation of the A2C performs better than A3C [OpenAI (2017)].

Because of a non-concave shape of the function to optimize, the policy could become deterministic too quickly. To overcome this problem researchers commonly use entropy regularization which allows to explore more the environment [Williams and Peng (1991)], [Ahmed et al. (2018)].

5 Experimental setup

Two different agents have been developed: $agent_{base}$ and $agent_{cnn}$. $agent_{base}$ uses $state_{vector}$ as state representation whereas $agent_{cnn}$ uses $state_{image}$ (states were described in section 3.1).

5.1 Neural Network

$agent_{base}$ uses the following neural network:

- Gated Recurrent Unit (GRU) of length 8
- 2 fully-connected layers of 4 units followed by a rectifier nonlinearity

$agent_{cnn}$ uses the network described in [Mnih et al. (2015)]:

- 32 filters of 8x8 with stride 4 with the input image and rectifier nonlinearity
- 64 filters of 4x4 with stride 2 followed by rectifier nonlinearity
- 64 filters of 3x3 with stride 1 followed by a rectifier nonlinearity
- fully-connected layer of 512 rectifier units

Both networks have fully connected layer of 2 units followed by softmax activation for the actor and fully connected layer of 1 unit for critic.

5.2 Training

Agents have been trained for 2000 episodes; A2C has been implemented using for each episode 3 parallel processes. For the $agent_{base}$ and $agent_{cnn}$ the following loss has been used:

$$\delta_t = G(s_t|a_t) - V_\theta^\pi(s_t)$$

$$loss = -\sum_{t=1}^{T} ln\pi_\theta(a_t|s_t)\delta_t + \sum_{t=1}^{T} \delta_t^2$$

where T is the number of timesteps per episode, G is the expected return at timestep t, s_t is the state at timestep t, a_t is the action chosen at timestep t, V_θ^π is the state-value function and π_θ is the policy. Another version of $agent_{base}$ has been trained using also the entropy regularization, in this case another term is added to the loss and it becomes:

$$loss = -\sum_{t=1}^{T} ln\pi_\theta(a_t|s_t)\delta_t + \sum_{t=1}^{T} \delta_t^2 + \beta\sum_{t=1}^{T} H(\pi_\theta(a_t|s_t))$$

where β is an hyperparameter to control the strength of entropy regularization and H is the entropy.

The optimization has been performed using RMSprop optimizer [Tieleman and Hinton (2012)] because in the original paper on asynchronous methods [Mnih et al. (2016)] it produced good outcomes.

The hyperparameters used during the training are described in the following table:

where β is the aforementioned hyperparameter whereas γ is the discount rate used for computing the expected return.

episodes	processes	learning rate	β	γ
2000	3	0.01	0.01	0.90

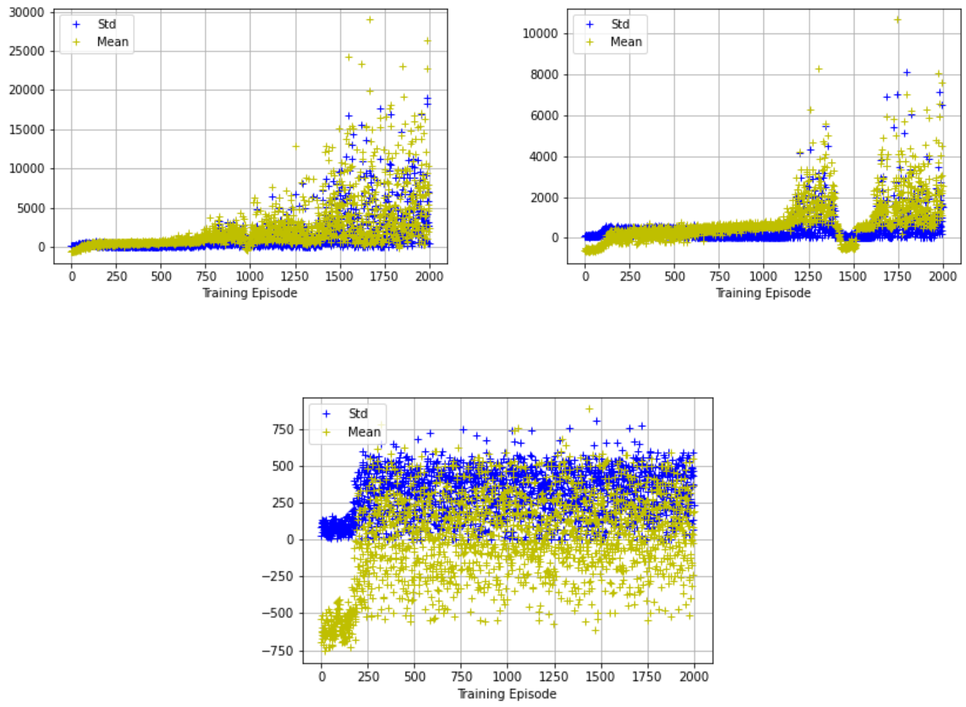

Figure 1: Average reward and standard deviation per episode, from left to right: $agent_{base}$ with standard loss, $agent_{base}$ with entropy regularization and $agent_{cnn}$ with standard loss

As shown in figure 1 the $agent_{base}$ learns well and the maximum average reward seems to have a soaring trend. Using a random initial state have both pros and cons: on the one hand it allows the agent to gather more experience exploring widely the environment; on the other hand it increases the standard deviation and causes still a low average reward in the last episodes. Maybe with more training episodes and more parallel processes this phenomena could disappear. $agent_{base}$ with entropy regularization seems to learn more slowly than the previous case as expected because the loss favors an higher entropy for the action choice to avoid to be stuck in a local optima. A peculiar trend seems to happen around the episode 1500 where the agent is no more able to pass through more pipes than before and learn another complete different policy. $agent_{cnn}$ shows disappointing result because the choice is very similar to a uniform random variable from episode 250.

5.3 Evaluation and Results

The aforementioned agents have been evaluated considering the score obtained in ten different games. Each score denotes the number of pipes the agent was able to pass through.

As shown in figure 2 the scores reflect the training results described in the previous section. $agent_{cnn}$ plays more or less randomly and this causes a lot of deaths above all at the beginning of the game where the agent touches the top or the bottom of the screen. The two versions of $agent_{base}$ (base and entropy) do not play badly even if for a small number of training episodes the agent trained without entropy regularization performs much better than the other one.

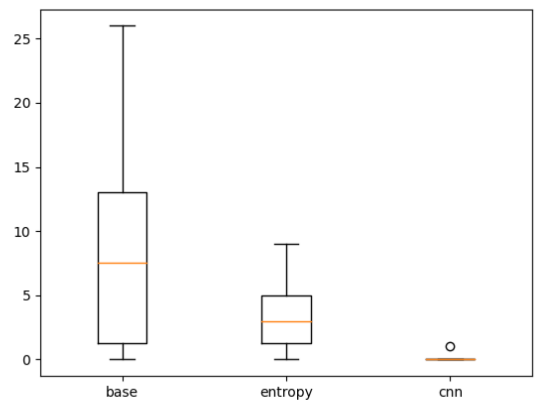

Figure 2: Boxplot of scores obtained by each agent during ten different games

6 Conclusions and future work

In this paper it has been discussed how A2C can be used to train an agent to play Flappy Bird. The representation of a state as a stack of images seems not to be the best choice in this case where a more feature oriented state turns out to provide better performances. The main problem of this work is the number of training episodes which is small compared to the amount of episodes needed to have better results. Differently from other kind of games, Flappy Bird cannot reach the convergence because the game never ends, therefore the best possible result is an increasing trend of the cumulative reward per episode. One of the main problems could be represented by the different behaviour the agent should have when it is outside or inside a pipe. In fact, the agent and the pipes are not points and segments but object with their own volume. For this reason when the agent must still pass a pipe there could be two different situations: it must reach the pipe or it is inside it. This is the case of the $agent_{base}$ trained without entropy regularization which is able not to crash into pipes but sometimes dies because it touches the bottom or the top of the pipes. Maybe this problem could be solved with more training episodes but it could represent an interesting point to investigate. Another aspect of the algorithm which could significantly improve the performance is the number of parallel processes which interact with the environment. In fact, in the original paper [Mnih et al. (2016)] 16 processes are used whereas in this case 3 processes need more interaction with the environment to get better results.

References

Ahmed, Z., Roux, N. L., Norouzi, M., and Schuurmans, D. (2018). Understanding the impact of entropy on policy optimization. *CoRR*, abs/1811.11214.

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. (2016). Openai gym.

Chen, K. (2015). Deep reinforcement learning for flappy bird.

Ebeling-Rump, M. and Hervieux-Moore, Z. (2016). Applying q-learning to flappy bird.

Eklavya Sarkar, M. C.-W. (2016). Playing flappy bird with deep reinforcement learning.

Gong, S. (2020). Introduction to reinforcement learning and q-learning with flappy bird.

Kovtun, M. (2019). Deep reinforcement learning for flappy bird using tensorflowjs.

Li, A. (2021). Reinforcement learning in python with flappy bird.

Lin, Y.-C. (2018). Using deep q-network to learn how to play flappy bird.

MacAdam, D. L. (1937). Projective transformations of i. c. i. color specifications. *J. Opt. Soc. Am.*, 27(8):294–299.

Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T. P., Harley, T., Silver, D., and Kavukcuoglu, K. (2016). Asynchronous methods for deep reinforcement learning.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. A. (2013). Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., and Hassabis, D. (2015). Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533.

OpenAI (2017). Openai baselines: Acktr & a2c.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., and Hassabis, D. (2016). Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489.

Talendar (2021). Flappy bird for openai gym.

Tesauro, G. (1994). Td-gammon, a self-teaching backgammon program, achieves master-level play. *Neural Comput.*, 6(2):215–219.

Tieleman, T. and Hinton, G. (2012). Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. coursera: Neural networks for machine learning, 4, 26-30.

Vu, T. and Tran, L. (2020). Flapai bird: Training an agent to play flappy bird using reinforcement learning techniques. *CoRR*, abs/2003.09579.

Williams, R. J. and Peng, J. (1991). Function optimization using connectionist reinforcement learning algorithms. *Connection Science*, 3:241–268.