Project DeepRL - Part 2

Clémence Grislain Julien Khlaut Aubin Tchoi Pierre Glandon

Introduction

The objective of this project is to achieve optimal performances in the game Flappy Bird. While the game mechanics in this version differ slightly from the original game, the basic premise remains the same: navigate through obstacles by passing above or under them. However, instead of the usual pipes with holes, this version features bars with varying heights and widths. Additionally, the game's handling of speed is different; whereas in the original game, performing an action always causes the character to ascend regardless of their speed, in this version, performing an action adds a boost to their current speed. As a result, the character may still be falling after jumping if they were already falling, and it is also possible to build up speed to the top of the screen. This increased inertia compared to the original game makes it more challenging.

To tackle this problem, we considered two approaches. The first approach involves using a tree-based algorithm that does not require the use of a deep neural network, and therefore does not need to be trained. This approach has the advantage of being computationally efficient, which is crucial since we only have access to two hours of Colab GPU training. The second approach employs neural networks, which we subsequently attempt to optimize by increasing its training speed.

Tree-based method

Motivation and outline

The initial motivation that supports this tree-based method comes from the observation that the dynamic of the problem is intuitive from a human's perspective. Notably, the horizontal velocity of the bird is constant, its vertical velocity is decreased each step by a constant that refers to the gravity and jumping increments this vertical velocity by a fixed value. The physics of the game is thus governed by three parameters: the horizontal speed, the vertical speed and the amount of speed granted by a jump. All three parameters are easy to infer if needed.

By reverse-engineering the environment, perfectly simulating a trajectory is thus within reach. The tree-based method is entirely built on this observation, and it operates on two parts: first we build a binary tree that models the boolean outcomes (losing or not losing) that result in all possible successions of actions (binary because there are two possible actions), and then we select the next action based on the predicted outcomes.

Implementation of the tree

For a given observation, when fed the three parameters mentioned previously, a tree is built recursively by performing two steps: computing the positions and vertical velocities that will be obtained when performing each of the two actions, and then checking whether the resulting positions will crash into an obstacle, including the top and bottom borders of the screen. Each of the leaves of the tree thus directly informs us on the outcome of one specific trajectory. The depth of the tree is chosen accordingly to the number of steps required to go past the last known bar: if we denote by $\mathcal{B}^r$ the right-end sides of the bars, by x the abscissa of the bird and v_x the horizontal speed, the depth d^* is computed as: $d^* = \lceil (x - \max_{b_x \in \mathcal{B}^r} b_x)/v_x \rceil$. Our implementation allows for setting a maximum number of bars to consider in order to limit the depth of the tree.

This structure allows for many optimizations, for instance if the bird crashes at some node of the tree, there is no need to compute the children of this node as any resulting leaf will bear a negative outcome. In its current implementation the positions and velocities on each node are actually not stored in memory, only the outcomes on the leaves are stored in a array of booleans, which is very light on memory. Storing the nodes would allow for tree recycling between each successive step by keeping the half of the tree corresponding to the action taken and updating it by only considering the changes induced by the apparition of a new bar. However, this approach would require storing the positions of all the bars in the nodes. Indeed, in the environment the bars move rather than the bird to avoid having values that tend to infinity, therefore in order to append nodes to the tree we would need to replicate this logic and store the corresponding information. Instead of 2^{d^*} leaves with boolean values, we would end up with $2^{d^*+1} - 1$ nodes, each of them storing a float (vertical velocity), a list of bars and a boolean. The balance in this trade-off between mem-

ory use and algorithmic complexity tilts in favor of using an array of booleans if we additionally take into account the handiness of working with a single array of boolean with a clear indexation.

Action sampling method

At each step, an action has to be selected accordingly to the outcomes computed within the tree. Rather than considering specific trajectories, we opted to see the tree as an indication of the number of trajectories that do not result in the bird crashing starting from any observation, which can be formalized as: $\mathbb{P}[\neg L|x_t, y_t, v_{y,t}, \mathcal{B}_t]$, where $\mathcal{B}_t$ is the set of bars observed at time t and L the event of losing. More specifically, the proportion of leaves with a negative outcome in the two halves of the tree yields the probability of losing when taking the action corresponding to the selected half first, and then performing random actions. Our method to select actions is therefore to take the action whose corresponding node has the most descendants with a positive outcome:

$$a_t = \arg\min_{a' \in \{0,1\}} \mathbb{P}[\neg L|f(x_t, y_t, v_{y,t}, a'), \mathcal{B}_t] \quad (1)$$

Where f is a function that updates (x, y, v_y) accordingly to the physics of the game and the action selected. This approach models the following motivation: to the best of our current knowledge, what is the action that offers the most breathing space to the bird in order to give him room to further dodge incoming bars. We can note that if there is only one possible trajectory, iteratively using this selection method will yield said unique trajectory.

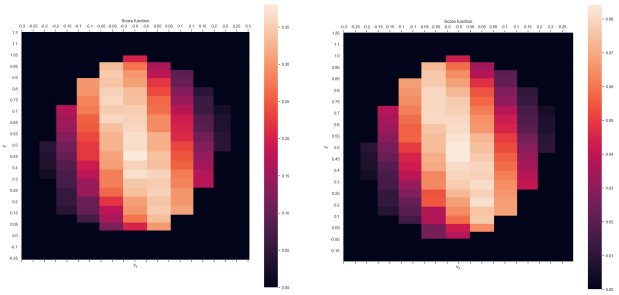
The approach described above is justified by the assumption that the risk induced by the random apparitions of bars roughly follow the same behavior in both halves of the tree. More precisely, it assumes that these new bars will reduce the number of viable trajectories uniformly throughout the discrete space of possible values for (x, y, v_y):

$$\begin{aligned} &\mathbb{E}_{B'}\big[\mathbb{P}[\neg L|f(x_t, y_t, v_{y,t}, 1), \mathcal{B}_t \cup \mathcal{B}'] \\ &\quad -\mathbb{P}[\neg L|f(x_t, y_t, v_{y,t}, 1), \mathcal{B}_t]\big] \\ &\approx \mathbb{E}_{B'}\big[\mathbb{P}[\neg L|f(x_t, y_t, v_{y,t}, 0), \mathcal{B}_t \cup \mathcal{B}'] \\ &\quad -\mathbb{P}[\neg L|f(x_t, y_t, v_{y,t}, 0), \mathcal{B}_t]\big]. \end{aligned} \quad (2)$$

This is probably not the case, as positions centered around the middle of the screen with low velocities intuitively offer more room to dodge upcoming bars and are therefore safer. To adapt our method to fit this intuition, instead of considering boolean outcomes we opted to enrich the model with a per-leaf score that is null for negative outcomes and decreasing with regard to some distance to $(y^*, v_y^*) = (\frac{1}{2}, 0)$ otherwise. We compared the performances of two parameterized scores:

$$\begin{aligned} s_{\text{convex}}(y, v_y) &= 1 - \alpha\sqrt{\beta(y - \frac{1}{2})^2 + (1-\beta)v_y^2} \\ s_{\text{geo}}(y, v_y) &= 1 - \frac{1}{c(\alpha, \beta)} \cdot |y - \frac{1}{2}|^\alpha |v_y|^\beta \end{aligned} \quad (3)$$

We introduced an additional score function, that perfectly accounts for the vertical asymmetry of the problem and for its exact dynamic. This score function re-utilizes the tree structure by computing a tree starting from each leaf without considering any of the bars. The depth of these per-leaf trees is fixed, and the only negative outcomes it contains come from the bird crashing on the top or bottom border of the screen. The proportion of leaves with a positive outcomes for a given tree yields the value of the score function on the corresponding leaf. As shown by the figures below, the score is naturally lower for leaves that are near the top (resp. bottom) border with a high positive (resp. negative) velocity (the figure shows y in rows and v_y in columns).



a) b)

Figure 1. Score function for various tree depths a) 10 b) 20

We observe that increasing the tree depth causes the score values to vanish, which is expected since deeper trees will extends many trajectories beyond the borders of the screen. This score functions allows for some leaves to have a contribution several times lower than some others. To balance out this effect, we added a constant term α to this score function: $(\alpha + s_{\text{exact}}(y, v_y))\mathbb{1}_{\text{positive\_outcome}}$. If $\alpha \to +\infty$, we recover the binary score in the comparison of the sums of the scores on the two halves of the leaves.

Results

To fasten up the cross-validation performed on the parameters α and β of the regularization functions, the experiment was parallelized on multiple threads. The tables in 2 show the results of the cross-validations for both models, averaged over 10000 episodes for episodes of length at most 1000 steps. We found that using fewer experiments led to results that were sensible to rare events and therefore lost statistical meaningfulness. Results are expressed in terms of number of steps reached, we are indeed mostly

interested in checking whether the agent successfully reaches the maximum number of step. The total reward is a less accurate indicator of the performance of the agent, in cases where the maximum number of steps is reached it is subject to random variations depending on the positions of the sampled bars at the end of the simulation. Execution times were independent of the parameters and of the score function chosen, with the exception of s_{exact}, which was optimized by caching the values of the score function. The bird indeed often takes the same positions (y, v_y), and through this memoization we recover the same speed as with the other two scores after only a few iterations.

The binary score ($\alpha = 0$ in the convex score) outperforms the geometric score by a small margin, which overall outperforms the convex one. Having a very high speed is indeed more of an issue if we are close to the border of the screen and vice-versa: the risks posed by two distances are relative to one another. The geometric score does not correctly consider the fact that the bird should have a high positive speed if it is close to the bottom and a high negative speed if it is close to the top. To take into account this interaction we conducted a series of experiments without the absolute values around $y - \frac{1}{2}$ and v_y, without any significant improvement.

An additional cross-validation over 10000 steps that ran for 46799 seconds (approx. 13 hours) was performed on the parameter α of the last-mentioned heuristic. The results obtained for $\alpha \in \{0, 0.2, 0.3, 0.5, 1.5, 2\}$ were all in the range $[991.60, 993.81]$ with the lowest standard deviation observed being equal to 66.94. The lack of discriminative power in this experiment does not allow for the selection of an optimal parameter, and for the sake of simplicity we are going to consider the binary score (ie. without regularization) as our best one overall.

Another metric available was the success rate, where a successful episode is one where the maximum number of steps is reached. This metric is interesting because it is agnostic to the randomness of the time within the episode of the apparition of extremely hard (and rare) configurations, where bars are set up in a way where it is nearly impossible to gain enough speed between two corridors. The average number of steps and the mean reward are sensible to it because such configurations will randomly appear soon in some episodes, and later in some others. However, it does correctly discriminate the ability of an agent to solve configurations based on how soon they occur in an episode, which would be useful if our agent did systematically fail on a certain type of configurations. The environment is indeed only partially auto-similar, the configuration of the bars at some point is mostly independent of the configuration found a certain number of steps earlier, but there is some border effect to consider because some configurations are of a certain size and thus cannot appear right at the beginning of an episode. Therefore, the success rate becomes more relevant as a metric for high maximum numbers of steps and then requires less episodes to average on. That said, the ranking of the different methods does not vary depending on the metric used.

The table below lists the results obtained in average over 100 episodes for episodes of length at most 1000 steps with a binary leaf-score.

	95% CI	Minimum	Maximum
Reward	348.54 ± 27.65	299	387
Steps	998.65 ± 26.33	865	1000

Table 1. Final results obtained with the tree-based method, refer to 3 for complete histograms

Note that even the worst-performing run gets the "Winner Winner Chicken Dinner" message and only one run ends before reaching the maximum number of steps. The run that did so ended in a configuration where all possible trajectories ended up with the bird crashing. The only margin of improvement thus lies in the anticipation of such configurations. Instead of the heuristic approach that utilizes various forms of leaf-score to guide the bird towards safer positions, we considered modelling the stochasticity of the apparitions of the bars in order to incorporate a prior to approximate the expectancies in (2). We did not pursue in this direction as the method used to sample bars did not seem trivial and was not reasonably reverse-engineerable. The heights (resp. vertical positions) of the bars seemed to be uniformly sampled in $[0, 0.6]$ (resp. $\{\text{UP}, \text{DOWN}\}$) but the positions of the bars on the x-axis seem to be drawn using a more complex law.

Deep learning-based method

For our deep learning solution, we employed the Deep Q-Network (DQN) as our agent. However, the environment poses an interesting challenge for training DQN algorithm, as a state is a tuple of information that includes both the bird's position and speed, as well as the shuffled information of the bars: their positions, height, and whether they are on the top or on the bottom of the screen. Based on this information, we attempted to design several rearranged states for our DQN in order to optimize its performances.

States

We have considered several way to encode states:

- **Vectorizing:** bird's coordinates, speed, and the bars' coordinates and heights, along with a boolean value

indicating whether each bar was on the top or bottom. We allowed for an undetermined number of bars, resulting in a larger vector that we padded with zeros to maintain a fixed size.

- **Single image:** reconstruction of the image of the game, assigning the bird and bars identical or different values.
- **Multiple images:** two reconstructed images, one containing only the bird and the other containing only the bars.
- **Tuple(image, velocity):** tuple combining a reconstructed image with the vertical velocity.
- **Sequence of images:** a series of five reconstructed images from the previous observations, with the option to separate or combine them, produced the best results.

The first method to encode states performs poorly. The second and third representations underperformed due to the lack of information on the velocity. The fourth state representation performed reasonably well, achieving an average score of 30 with a well-designed neural network. The fifth state representation yielded the best results, but reconstructing everything on the same image with different values for the bird and the bars proved just as effective as separating them into two images, although the single-image approach was significantly faster. With this state representation and an optimized neural network, we were able to achieve a score of 200 within two hours of training.

Network

We employed two distinct neural network architectures. For the tuple state representation, we utilized a network comprising 2D convolutional layers on the input image, followed by flattening the resulting tensor and concatenating the bird's speed as an additional feature. We then passed the vector through a MLP with a hidden state of 256 neurons, enabling the network to learn complex relationships between inputs and outputs.

Alternatively, for the multiple images state representation, we designed a network that utilized 3D convolutional layers to process the input images, allowing the network to learn the bird's speed on its own. We then flattened the tensor and fed it through an MLP with a hidden state of 256 neurons.

In both cases, we employed a learning rate of 10^{-4} to optimize the model's performance.

Speeding up the trainig with a pre-training

To accelerate our learning process, we implemented two different methods. The first was inspired by the observation that our agent was taking longer and longer to train as it improved, and was specifically struggling when encountering challenging combinations of bars that created shallow paths. However, these passages occurred infrequently during training, hindering the agent's ability to learn from them. We attempted to address this by complexifying the environment as the agent learned. By increasing the height and probability of upcoming bars every N epochs, we hoped to provide the agent with a richer training environment. However, we found that increasing height beyond 0.6 often resulted in nearly impossible bar configurations, limiting the effectiveness of this approach. Instead, we focused on increasing only the probability of the next bar, resulting in faster training times (7000 epochs to reach 100 reward, down from 15000) by allowing the agent to experience and learn from more challenging combinations.

We also found that our agent struggled to reach high performance when training because the exploration parameter ϵ was set too high at 0.1, preventing the agent from achieving higher rewards. We also periodically trained the agent on the actual game environment, leading to drastic improvements in performance.

Through these techniques, we were able to accelerate our reinforcement learning process and achieve better performances in a shorter amount of time.

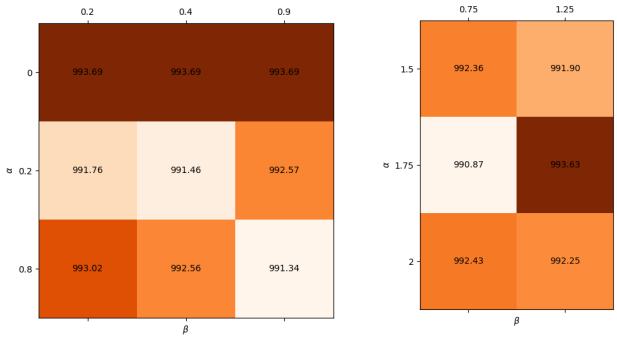
Conclusion

To solve this game of FlappyBird, we developed several methods, deep-learning based ones and deep-learning free ones. With the limitation enforced on the maximum training time allowed, our implementation of a DQN agent fell short in the competition and was outperformed by our custom tree-based algorithm.

When it comes to training a neural network, the specificity of the game that makes it hard to learn lies in the inertia of the bird, that creates the need to account for a distant future very precisely and makes it almost impossible to quickly recover from a series of poor decisions. Our pre-trained DQN agent also proved to be inherently unstable, each pre-training leading to agents of various performances. In comparison, our tree-based agent succeeds in losing only a handful of times over hundreds of episodes of length 1000.

To further improve the performances of this agent, we tried incorporating a prior to guide the bird towards what we believe to be safer positions. We introduced three distinct heuristics governed by a few parameters that we intensively cross-validated. The most sophisticated one involves computing a full tree to account for the probability of loosing when choosing random trajectories starting from a given position and velocity. This work can constitute a baseline for problems with a small set of decisions and with a fixed and reverse-engineerable dynamic.

Appendices



a) b)

Figure 2. Cross-validation results for a) s_{convex} b) s_{geo}

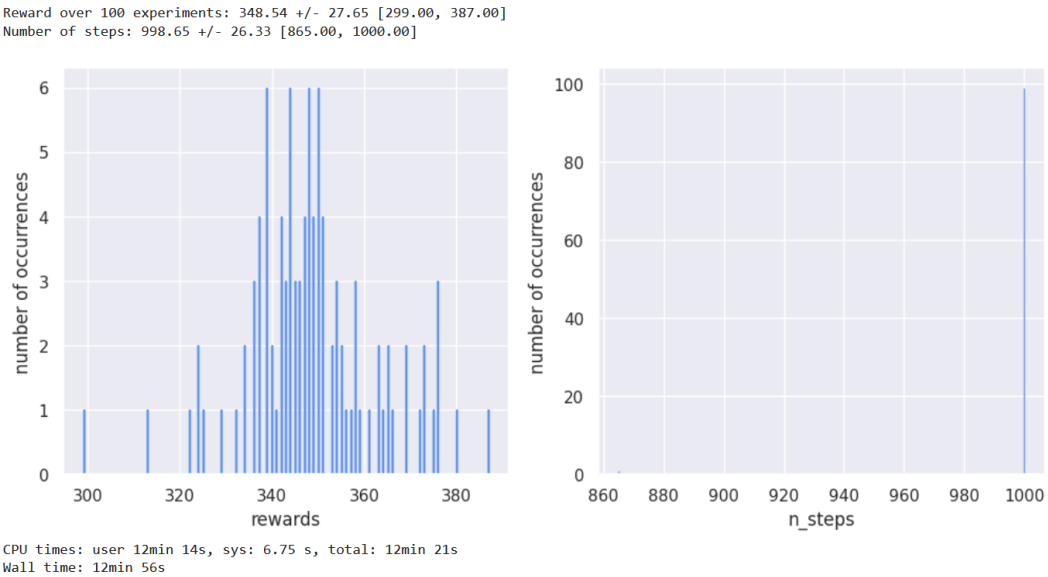


Figure 3. Histograms of the rewards and number of steps obtained with the tree-based agent

Note that in the histogram of the number of steps, there is a single bar at 865 steps of height 1 other than the one at 1000.

α	0	0.2	0.3
Steps	991.60 ± 146.72	992.38 ± 134.25	991.74 ± 151.20
α	0.5	1.5	2
Steps	993.81 ± 135.97	992.62 ± 137.18	992.79 ± 131.21

Table 2. Cross validation results for $\alpha + s_{\text{exact}}$