Enhancing Decision-Making in Web Games Through Reinforcement Learning

Ronit Chougule[1](✉) and Anushka Nuzhat[2]

[1] School of Computer Engineering and Technology, Dr. Vishwanath Karad MIT World Peace University, Pune, India
1032200841@mitwpu.edu.in

[2] Department of Electrical and Computer Engineering, North South University, Dhaka, Bangladesh
anushka.nuzhat@northsouth.edu

Abstract. Decision making is an essential attribute of any intelligent agent. In this project, we addressed the problem of teaching an agent to learn the optimal policy for a given state. In context of the web games presented below, we taught both the doodle in Doodle Jump, and the bird in Flappy Bird, a set of moves that maximized their respective long-term cumulative score (a.k.a. reward) by utilizing the subset of Machine Learning algorithms known as Reinforcement Learning. We explicitly defined the three main components of a reinforcement learning algorithm—the state set, the action set, and the rewards—in such a manner that convergence was relatively quick, and performance was above that of a skilled player. Our results indicate that there exists a trade-off between the granularity captured by the state set and how long it takes until the algorithm performs well. In Doodle Jump, a much smaller state set resulted in faster learning, but lower absolute scores when compared to a higher granularity variant (avg score of 8000 versus 23,000). High granularity variants performed well in the long-run, but took significantly longer to train and required more resources for the algorithm to work. By providing this report (Maddison et al., Move evaluation in go using deep convolutional neural networks, 2014; Mnih et al., Playing atari with deep reinforcement learning, 2013), we hope that we can improve the effectiveness of Reinforcement Learning algorithms in fully-fledged and modern web games with non-trivial state-action sets.

Keywords: Reinforcement learning · Web games · Decision making

J. Lu et al. (Eds.): IRC-SET 2023, *Proceedings of the 9th IRC Conference on Science, Engineering, and Technology*, pp. 95–105, 2024.
https://doi.org/10.1007/978-981-99-8369-8_10

1 Introduction

In other Supervised Learning algorithms, specific samples with explicit labels are given as input with the objective of recognizing patterns within the data to provide a meaningful output [1]. However, in Reinforcement Learning, there is no predefined correct output to a specific input, but rather there exists an agent that learns the best possible output for a given state through trial and error [2].

A game is a simple and efficient way to see the effects of Reinforcement Learning. In most games, an agent, ideally, performs an action in each state that increases its long-term score, where the states are defined as a combination of any factors that affect the agent's decision [3]. As the agent explores the state set, it rewards actions that increase its score and penalizes actions that do not, thus allowing the agent to "learn" [4].

The problem of building intelligent agents in games has been addressed by many algorithms in the past. For example, Deep Convolution Networks have learned how to play Go [5], and Deep Q Networks have learned how to play over 49 classic Atari Games to the same level as professional human game testers [6]. However, not a whole lot of progress has been made towards applying reinforcement learning algorithms to popular and modern web games. While the implementations referenced above are important, a novice machine learning enthusiast would more likely benefit from seeing how to apply reinforcement learning algorithms to modernly-implemented web games, which have been gaining popularity over their desktop application counterparts recently [7].

In this paper, we show an effective implementation of the reinforcement learning algorithm called Q Learning to Doodle Jump, after being inspired by the performance of the same algorithm applied to Flappy Bird, a much simpler, but still immensely popular game [8]. We also introduce new factors to the both games: granularity/resolution, exploration, decision frequency, and tick frequency, which help reduce the amount of time it takes for the algorithm to perform well. A combination of the new and old hyper-parameters of Q Learning show scores of over 30,000 (in DJ)—well above that of a skilled player—in a relatively short amount of time without much computation power.

2 Methods

2.1 Flappy Bird

Input:Local

- S represents a collection of states
- Q[s, a] i.e. the real array
- A denotes a set comprising actions
- s i.e. the prior state
- γ denotes the discount coefficient
- a i.e. the prior action
- α represents the size of step.

Pseudocode

Initialize Q[s,a] arbitrarily

Observe the current state s

Repeat

Select and carry out action a

Observe reward r and state s′

$$Q[s,\ a] \leftarrow Q[s,\ a] + \alpha\left(r + \gamma\ \max_{a} Q[s\prime,\ a\prime] - Q[s,\ a]\right)$$

s ← s′ until termination

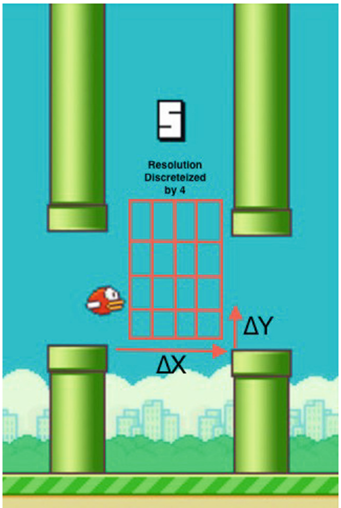
$$Q(s, a) = Q(s, a) + \alpha\left(reward + \gamma Q\left(s', a'\right) - Q(s, a)\right)$$

Application of Q Learning to Flappy Bird: The application of Q Learning to Flappy Bird is training an agent, the bird, to perform an optimal action in any state. In order to accomplish this, the agent explores a state set and assigns rewards based off whether an action from one state produced a favorable outcome [8]. Q Learning is a model-free form of reinforcement learning [9].

Learning: The array Q is initialized with zeros. Every time we reach a state, denoted by *s*, we choose the best action (the action that will maximize the expected reward). In the case of a tie, we "Do Nothing."

1. Observe the state s and perform the action *max Q*[*s*, *a*] that maximizes the reward. On the next tick of the game engine, the agent is now in the next state s'.
2. Observe the new state, s' and the reward that is associated with it. If the bird is still alive + 1, otherwise − 1000.
3. Update the Q array according to the rule we defined in the previous section.
4. Set the current state to s' and repeat the loop.

State Set Reduction The state set is defined by the vertical distance from the next lower pipe to the bird, ΔY, and the horizontal distance from the next pipe to the bird, ΔX. Due to the continuous nature of the state set, there is a division to both the vertical and horizontal distances called the *resolution/granularity* factor. This means that the result is rounded to the nearest integer to make the state set discrete and small.



Reward Process The agent was given a reward at every tick, which occurs at the rendering of each frame. If the agent died as a result of their choice, we assigned the action selected a reward of -1000. If the agent lived, we assigned a small reward of 1. By doing this, we effectively penalized deaths, resulting in the agent learning how to maximize its life, and therefore its score.

Action Process In this game, the agent can only perform two actions in any given state: jump up or do nothing and begin to fall.

2.2 Doodle Jump

Parameters

- *xdivision, ydivision*: Round distance to the nearest division
- *brain.learning rate*: Scales how fast the predicted rewards change
- *scale death*: Adjusts the penalty for death at higher scores
- *decision*: The y velocity at which a decision is made.

Local Variables *previous score, previous collision, target platform, last state*: info from last decision

actions: The cumulative reward for going to each platform.
explored: The number of unique states encountered.

Implementation

1. The "brain" of the agent is a 3-dimensional array of cumulative rewards indexed by the state of each on screen platform.
2. The states are a 3-tuple of platform type, y-distance, and x-distance. The state space for our model is 2580.

(a) Platform type takes on the value of 0 for solid platforms, 1 for broken plat-forms, and 2 for moving platforms.
(b) Y-distance is the vertical distance between the player and the platform ranging from − 550 to 310, rounded to the nearest *ydivision.* This distance is relative as it is important to know if the platform is above or below.
(c) X-distance is the horizontal distance between the player and the platform ranging from 0 to 400, rounded to the nearest *xdivision.* This distance is absolute as the direction does not affect the possibility of landing on it.

State Set Reduction Using floating point for the position of the platforms and the player would result in over 1 billion possible states. Thus, it was important to reduce the dimensionality of the problem. The x distance and y distance are rounded to integers. However, simply rounding to the nearest integer would still result in over a million states. The parameters *xdivision* and *ydivision* are used to round the distances into a small set of discrete values. Larger divisions result in a smaller state set and decreased accuracy. Through trial and error, it was determined that the values *xdivision* = 40 and *ydivision* = 10 resulted in a sufficient compromise between model complexity and accuracy. The platform height is 17, the width is 70, so these values provide sufficient overlap and granularity. Figure 1 shows the effect of *xdivision* and *ydivision.* Figure 1a shows two platforms that would be recognized as different states in our model. Figure 1b–d show pairs of platforms that would be recognized as the same state in our model. With lower division values, these could be distinguished at the cost of increased complexity, training time, and computing requirements.

Fig. 1. Platform granularity.

Decision Process

1. At a predetermined agent velocity, set by the variable *decision*, read in all possible next states.
   (a) If the state has been seen before, then it exists within the brain object.
   (b) If the state has never been seen before, add the state to the brain object and initialize its reward to a random number from 1 to 100.
2. Pick the platform with the maximum reward based on the value indexed by a state in the brain object.
3. Assign this as the target platform, then move towards it.

Reward Process

1. If the agent moves towards the target platform and dies, penalize that platform's state by a fixed amount.
2. If the agent moves towards the target platform and successfully lands on it, reward that target platform with the increase in score.

3. If the agent moves to the target platform, but does not successfully land on it and does not die, then reward the platform the agent landed on with the increase in score and penalize the target platform by a fixed amount. Penalizing the missed platform causes the doodle to change its decision if it gets stuck.

3 Results

3.1 Flappy Bird

Here, X axis = trials and Y-axis = scores.

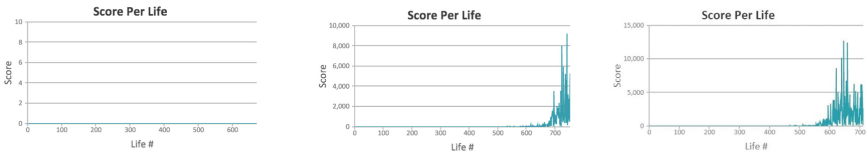


Fig. 2. Varying α, the learning rate.

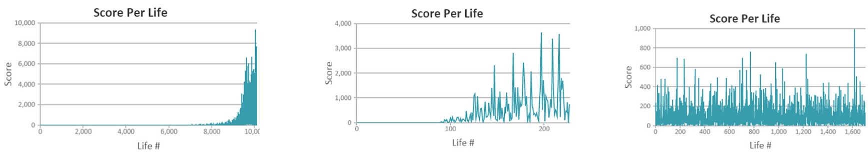


Fig. 3. Varying resolution/granularity, the state set size divider.

Here, X axis = trials and Y-axis = scores.

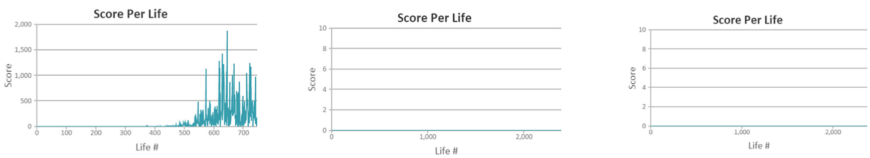


Fig. 4. Varying ε, the exploration rate.

Here, X axis = trials and Y-axis = scores.
Here, X axis = trials and Y-axis = scores.

3.2 Doodle Jump

See Tables 1 and 2.

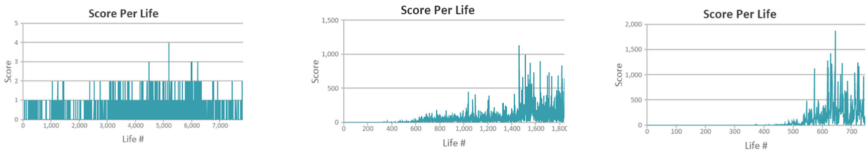


Fig. 5. Varying γ, the discount rate.

Table 1. Varying granularity.

X division	8	20	40	60	80	100
Y division	2	5	10	15	20	25
Learning rate	1	1	1	1	1	1
Avg score	5000	23,000	12,000	15,000	28,000	8000
States explored	22,902	5389	1745	947	600	419
Total states	64,500	10,320	2580	1148	644	426
States explored %	35.51	52.22	67.64	82.49	93.17	98.36

Table 2. Varying learning rate

X division	40	40	40	40	40	40
Y division	10	10	10	10	10	10
Learning rate	0.1	0.25	0.5	0.75	0.9	1
avg score	16,000	13,000	15,000	11,000	10,000	12,000
States explored	1745	1733	1746	1741	1719	1745
Total states	2580	2580	2580	2580	2580	2580
States explored %	67.64	67.17	67.67	67.48	66.63	67.64

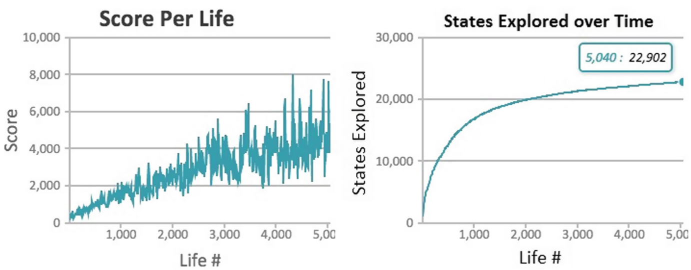


Fig. 6. Model with *xdivision* = 8, *ydivision* = 2, *learning rate* = 1

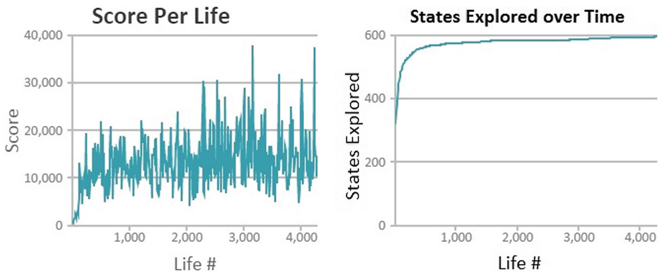


Fig. 7. Model with *xdivision* = 80*, ydivision* = 20*, learning rate* = 1

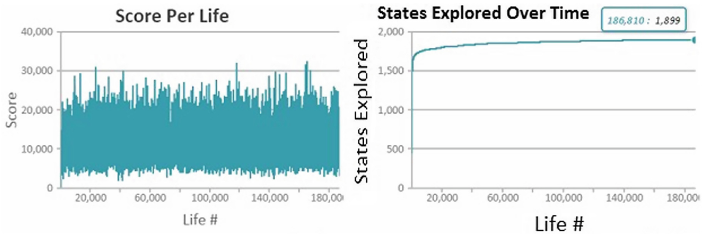


Fig. 8. Model with *xdivision* = 40*, ydivision* = 10*, learning rate* = 1

4 Discussion of Results

4.1 Flappy Bird

We sped up the tick rate of the game in order to reduce the amount of time it took to converge from 6 to 7 h down to 15 min. Additionally, all four of the algorithm's hyper-parameters were exposed for the user to adjust and gather data. Different combinations of these parameters, α, *Resolution/Granularity*, *Exploration Rate* ε, and the Discount Rate γ, produced the results itemized in the following subsections. Certain combinations of these parameters produced satisfactory results, often yielding scores over 9000.

Varying α, the Learning Rate (refer Fig. 2) With alpha set to 0 and the rest of the variables constant, the score was constantly zero. At $\alpha = 0.2$ the bird starts increasing its score after the 600th trial, scoring 8000 after the 700th iteration. With $\alpha = 0.6$, the score starts increasing significantly after the 500th iteration, reaching well above 10,000 around the 650th iteration.

This observation makes sense since the learning rate is essentially how much of the newly discovered information is stored [10]. Typically, we would start with a high learning rate, which allows fast changes, and then we would slowly decrease it as time progresses [1]. So, when α was 0, it never retained any new information it learned, thus its score was stuck at zero. This coincides with what we would find if we set α to 0 in the Q-Learning equation presented in Sect. 2.1.

Varying Resolution/Granularity, the State Set Divider (refer Fig. 3) The value of resolution is significant in that it decides the size of the state set. The *floor()* function is utilized to help us divide the game screen into discretized states. In our 540×180-pixel screen, if resolution is set to 10, the size of the state set would be determined as follows: *floor(540/10)* $= 54$*; floor(180/10)* $= 18$. The size of the state set is therefore 54-by-18. The higher the resolution, the smaller the state set, and vice-versa. A larger state set results in more accuracy at the expense of additional training time and computational resources. A smaller state set takes less time to train, but results in less accurate decisions being made since, in this particular case, states are grouped together with adjacent states.

By setting α to 0.7, Exploration Rate to 0, γ to 1, and letting the Resolution divider vary, we show the balance between accuracy and performance (Fig. 3). We found that when $Resolution = 1$, the Q-Learning algorithm learns very slowly and only gains any sort of score after approximately 7000 deaths; however, it reaches a maximum score around 10,000. When $Resolution = 10$, it starts to make intelligent decisions after approximately a couple hundred deaths and reached a maximum score around 4000. When $Resolution = 100$, the algorithm gained score as soon as it began but it can only reach a maximum score of < 1000.

Varying ε, the Exploration Rate (refer Fig. 4) In the context of Q Learning, the Exploration Rate determines the "willingness" of the algorithm to try a random action, completely ignoring what it has learned, in hopes that this random action will result in better long-term performance [11]. When the exploration rate is set to zero, the bird is conservative and always picks the best-known action for the given state. However, given that there are only two actions that can be taken in any state (up, down/do nothing), any relatively high value for exploration rate would prevent the bird from truly learning. A random action taken in such an instance is binary and is thus not complex enough to truly utilize the randomness of the exploration rate. This becomes very apparent in the two figures provided in Sect. 3.1 (Fig. 4). With exploration rate set equal to 0.4 and 0.8, the bird does not learn at all with an average score of zero.

Varying γ, the Discount Factor (refer Fig. 5) The discount rate determines whether there is more emphasis on the current reward compared to future rewards. The results of the score per life plots convey that the higher the γ, the higher the score. From the graphs in Sect. 3.1 (Fig. 5), we can see that for γ equal to 0, the highest score for over 7000 trials was only 4. On the other hand, for γ equal to one, we reached a high score of roughly 1700 by the 700th trial. For an intermediate value of γ equal to 0.4, the highest score of roughly 1200 was not reached until the 1500th trial. Therefore, we can conclude that not only does a higher γ result in a higher score, but that it also reaches the higher score much faster. In other words, it learns better in this particular game.

Our results coincide with the game engine of the Flappy Bird and the Q Learning algorithm presented in 2.1. Since we force our agent to select the best action after each frame is rendered, our agent must heavily weigh long-term rewards over short-term ones. This is due to the large iteration gap between when a decision is made, and how it affects the agent in the long run, hence why our results show that a higher γ performs better than a lower one.

Summary of Results We have found, empirically, that a good hyper-parameter variant for Flappy Bird is one with an α near 1, a small state set divider, an exploration factor that starts at 1 and quickly goes down to 0, and a γ near 1.

4.2 Doodle Jump

Division Granularity The results for each of the experimental models were compared after 5000 iterations (Table 1). The absolute number of states were affected by varying the division sizes for x and y distances. Also, division size enhancement results in a lower number of total states, and a decrease in division size results in a higher number of total states. Therefore, reducing division sizes resulted in less states being explored in the same amount of time. This also introduced states that would never be explored, because the game has a minimum and maximum y-distances between platforms.

Learning Rate (Table 2) The experimental models used for learning rate were compared in the same manner as the experimental models for division granularity. The models had similar exploration rates. Lower learning rates resulted in larger scores because the agent was able to explore more options before the weight of other options grew. With a large learning rate, the agent quickly learns a few good options, and is less likely to explore other candidates.

Summary of Results Fig. 6 shows that with more divisions, it takes more time and space to train the model. Training this model to convergence would likely result in a slight increase in score. However, given computational and time constrains, the (40, 10, 1) model used in Fig. 8 provides adequate accuracy while taking much less time and space to converge. With the (40, 10, 1) model, the majority of states are discovered in the first 5000 lives. The average score also does not change after this point (Figs. 7 and 8).

5 Concluding Remarks

Our project involved showcasing the practical implementation of Q Learning in well-known web games such as Flappy Bird and Doodle Jump. We have described, in detail, the precise methodology implemented to teach the agents the optimal set of actions that maximize their respective cumulative score. After tweaking the hyperparameters, we observed how state sets and their sizes really determine both how well the algorithm performs, how long it takes until the algorithm performs well, and whether it will even converge given the resources available to the algorithm. With a significantly reduced state-action set, we have achieved results that surpassed that of an expert player in both games.

Given how we trained our agents, one might argue that they should never die after a sufficient amount of training. In theory, this is true. However, in practice, we suffer from the curse of dimensionality and the limits of modern computers. This limitation can be dealt with in the future by using parallel computing—i.e., running multiple games at a time and merging results from all [12]. By doing this, one would be able to significantly cut the time it takes for the algorithm to converge, even if the state set is large.

While we did not get that far in this project, we hope that our contributions can help others apply reinforcement learning algorithms to other popular web games, perhaps ones that have models much more complex than the ones presented in this report.

References

1. Nasteski, V.: An overview of the supervised machine learning methods. Horizons **4**, 51–62 (2017)
2. Barto, A.G.: Reinforcement learning control. Curr. Opin. Neurobiol. **4**(6), 888–893 (1994)
3. Danzi, G., Santana, A.H.P., Furtado, A.W.B., Gouveia, A.R., Leitao, A., Ramalho, G.L.: On line adaptation of computer games agents: a reinforcement learning approach. In: Proceedings of the II Workshop de Jogos e Entretenimento Digital, pp. 105–112 (2003)
4. Yang, Y., Wang, J.: An overview of multi-agent reinforcement learning from game theoretical perspective. arXiv preprint arXiv:2011.00583 (2020)
5. Maddison, C.J., Huang, A., Sutskever, I., Silver, D.: Move evaluation in go using deep convolutional neural networks. arXiv preprint arXiv:1412.6564 (2014)
6. Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., Riedmiller, M.: Playing atari with deep reinforcement learning. arXiv preprint arXiv:1312.5602 (2013)
7. Chen, T.T.: Online games: Research perspective and framework. Comput. Entertain. **12**(1), 1–26 (2015)
8. Ebeling-Rump, M., Kao, M., Hervieux-Moore, Z.: Applying q-learning to flappy bird. Department of Mathematics and Statistics, Queen's University (2016)
9. Dayan, P., Watkins, C.: Q-learning. Mach. Learn. **8**(3), 279–292 (1992)
10. Ray, S.: A quick review of machine learning algorithms. In: Proceedings of the 2019 International Conference on Machine Learning, Big Data, Cloud and Parallel Computing (COMITCon), pp. 35–39. IEEE (2019)
11. Kaisers, M., Tuyls, K., Parsons, S., Thuijsman, F.: An evolutionary model of multi-agent learning with a varying exploration rate. In: Proceedings of the 8th International Conference on Autonomous Agents and Multiagent Systems, pp. 1255–1256 (2009)
12. Li, L., Lin, Y., Zheng, N., Wang, F.Y.: Parallel learning: a perspective and a framework. IEEE/CAA J. Autom. Sin. **4**(3), 389–395 (2017)