Jurassic Parkour*

Reinforcement Learning in an HTML5 Browser Game

Sebastian Laudenschlager, Ian Martiny

1 Introduction

The goal of this project is to use reinforcement learning to train a computer agent to play the Chrome T-Rex game. To that end, we use data that is procedurally generated by running the infinite runner game in a Google Chrome Browser. The agent has a set of actions, and the environment has a set of states, and the agent will choose an action based on the state of the environment. To make decisions, act accordingly, and learn from the result, we utilized the REINFORCEjs library [1], which implements some reinforcement learning algorithms such as Deep Q-Learning. The platform we use to test the learning algorithm is the Javascript T-Rex game running on a locally hosted python server using Flask. We first discuss some of the previous related work that this project was built on, followed by a more detailed look at the methods and algorithms we used. Subsequently, we present some data from several experimental runs and comment on the performance of the agent. Finally, we provide some commentary on the successes and failures of this project, as well as some ways to improve on this project.

2 Related Work

Our work draws from varying motivations and uses multiple resources we'd like to acknowledge here. The two course provided sources [2, 3] were very helpful in learning the information necessary to use any reinforcement learning techniques.

We examined many different tools to use for our project. Initially we attempted to look at PyBrain [4], a `Python` library for reinforcement learning. While promising and accomplished we ultimately ran into issues attempting to do our learning on a `Python` server, while the game ran on a `JavaScript` client. Our original plan was to run the client and pass a score to a `Python` server which then uses a reinforcement learning library to compute new parameters. However with the nature of our game needing to update the "agent" with a procedurally generated environment required a much quicker and constant interaction between the game and the agent.

To this end we examined other projects which were based in `JavaScript` that used reinforcement learning. One in particular that we found useful was FlappyBirdRL [5]. Though it appears that this project has a self designed reinforcement learning library tailored specifically to their project. While interesting we were unable to adapt this to fit our needs.

Finally we examined REINFORCEjs [1], a `JavaScript` library for reinforcement learning. This fit our needs perfectly: a self-contained library, meant to interact with `JavaScript` programs directly.

*code at https://github.com/IanMartiny/jurassic-parkour

3 Methods

3.1 Baselines

Prior to testing the reinforcement learning algorithm, we established two baseline tests for the T-Rex game: (1) Random jumping and (2) Increasing Threshold.

The random jumping baseline consists of the python server computing a random threshold between 0 and 1 and sending it to the Javascript client. The client then computes its own random threshold value between 0 and 1, and then compares the two threshold values. If the client's value is larger, the T-rex jumps, otherwise nothing happens. We do this for every run, meaning that once the T-Rex "dies", the game resets, and new values are computed. We also keep track of the score for each run, and then take the average over many runs.

The increasing threshold baseline works as follows:

- If the reported score is between 1000 and 2000 (inclusive) then the new threshold is increased by a tenth of the distance from 1. These are considered low scores. A score less than 1000 is not possible. This allows the threshold to increase by a significant amount, without going above 1.
- If the reported score is between 2001 and 4000 (inclusive) then the new threshold is increased by a fiftieth of the distance from 1. These are still relatively bad scores, but realistically the best we can hope for with purely random jumping. For this model these scores are considered good enough to not change much.
- If the reported score is anything higher the threshold is increased by a hundredth of the distance from 1. Any score above 4000 is considered very good for this model and thus the threshold should not change very much.

We again keep track of the scores for many runs, and then take the average of the scores.

3.2 Reinforcement Learning

To actually train a computer agent to learn to play the T-Rex game, we used a reinforcement learning algorithm, specifically the Deep Q-Learning Algorithm from the REINFORCEjs library. The agent essentially learns in the following way:

- First, we initialize the agent with a set of parameters including the learning rate α, the greedy policy ϵ, the learning algorithm, experience size, etc.
- We then start a loop to keep playing the game. At the beginning of every run, we reset the environment states.
- Based on the state of the current environment, the agent chooses and executes an action.
- Based on the result of the action, a reward/penalty is applied.
- The agent then updates its policy accordingly, and if the T-Rex "died", the game restarts. If the agent successfully avoided an obstacle, the environment is updated to reflect the next obstacle, and the process repeats.

The environment states at any point in the game are described by four variables:

- The current speed of the game, i.e. how fast obstacles are moving toward the T-Rex
- The height (y-position) of the T-Rex
- The horizontal distance between the nearest obstacle and the T-Rex

- The height of the nearest obstacle

The action space for the agent is (1) Idle (do nothing), (2) Jump, and (3) Duck. Due to the continuous updating of the environment states, we have continuous state features and discrete actions, which led us to using REINFORCEjs's **DQNAgent**.

The idea of Deep Q-Learning is to utilize a neural network in combination with Q-learning. In particular, the neural network will decide which action to take, and the loss function in the neural network will be updated using Q-learning, since we do not have a nice labelled outcome. The use of Deep Q-Learning then further expands our parameter space, as we need to consider parameters such as the number of neurons in the hidden layer.

Originally the rewards/penalties were assigned such that successfully jumping over an obstacle was heavily rewarded (e.g. reward of +50), "dying" was heavily penalized (e.g. penalty of -25), and being idle was very lightly rewarded (e.g. reward of +3). The intention behind this set of rewards and penalties was to avoid obstacles by jumping over them, and to do so without constantly jumping. Furthermore, we experimented with several combinations of these values to attempt to improve the rate of learning, the results of which are discussed in the next section.

As mentioned above we considered a few "features" for our agent to consider, we felt that these best encapsulated the necessary information for the agent to decide when and where to jump—how close (in the x direction) the next obstacle was, the height of the obstacle (whether to jump or duck), how fast the game was moving (how far away to start the jump/duck), and the current height of the T-Rex (if they're in the air, they might need to duck to avoid a bird).

4 Results

4.1 Baseline

As reported in our Baseline report, we analyzed how "effective" our baselines were. For our baseline models we had a server-client model, where the server would choose the threshold for the client to jump and the client would report the score to the server. For our first baseline we used the model where the server ignored the client's score and chose a new threshold randomly, resulting in the following results:

After recording the scores for 100 runs we found that the average score was 1850.045, this corresponds to slightly better than not jumping at all.

Our second baseline was a bit more complex, but not "intelligent". Essentially we started the client with a completely random (50%) chance to jump and then slowly decreased that based on how well the client performed (see Section 3.1) this baseline gave the following results:

After recording the scores for 100 runs we found that the average score was 1699.568, this corresponds to essentially not jumping at all. Due to the threshold always increasing, eventually the threshold is so large that the T-rex never jumps.

4.2 Reinforcement Learning

Our general methodology is discussed in Section 3.2, here we will discuss how our parameters were set and the results of those experiments.

For each experiment we stored data on how the agent performed every 50 trials, that is, in blocks of 50 trials we recorded how many successful jumps the agent performed, as well as the corresponding value of ϵ.

Our first experiment set our learning rate $\alpha = 0.005$, and our greedy policy $\epsilon = 0.20$ (and slowly decreased it every action taken). We then chose certain actions to be rewarded and

penalized as follows:

$$\begin{aligned} \text{idle reward} &= 3 \\ \text{success reward} &= 50 \\ \text{die penalty} &= -25 \end{aligned}$$

in short, we lightly rewarded our agent for choosing to be idle (neither jumping nor ducking), highly rewarded our agent for successfully jumping over an obstacle and heavily penalize the agent for dying (running into an obstacle). These results are displayed in Figure 1.

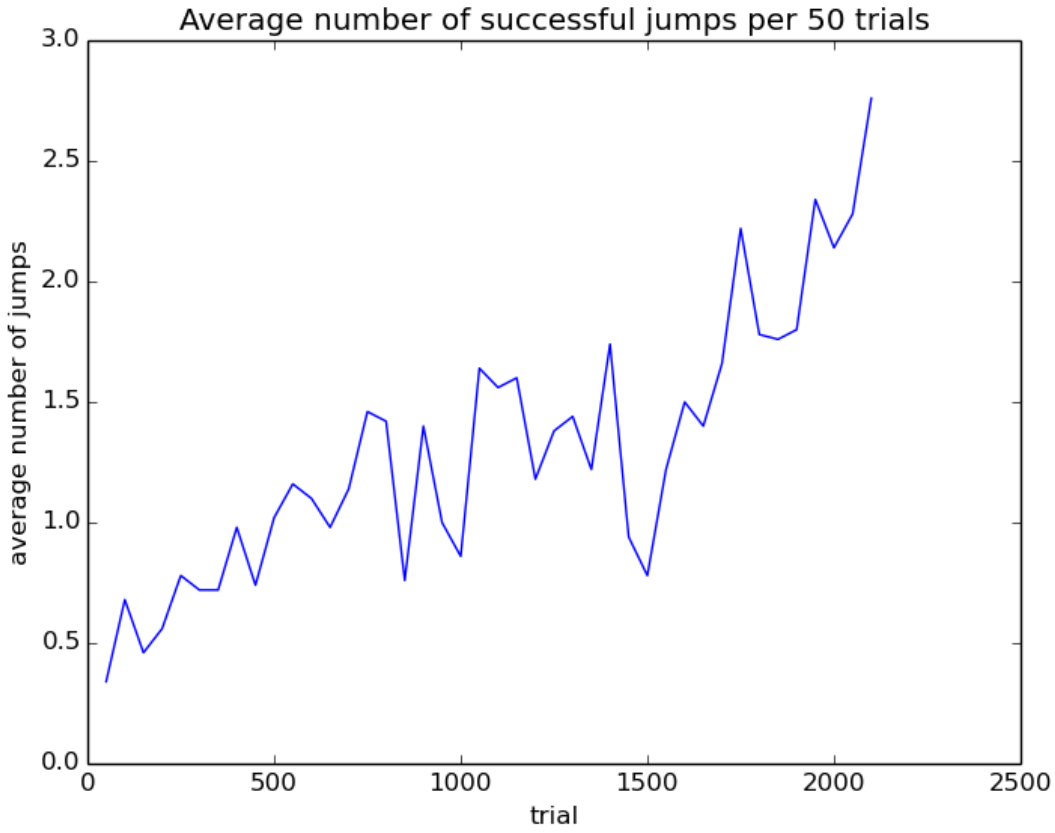

Figure 1: Average number of successful jumps with $\alpha = 0.005$, no jump penalty, idle reward = 3, success reward = 50, and dying penalty = -25

Our second experiment changed essentially nothing except that we imposed a jumping penalty to our agent, of -10, hoping to curb the agent's enthusiasm to continuously jump. These results are displayed in Figure 2.

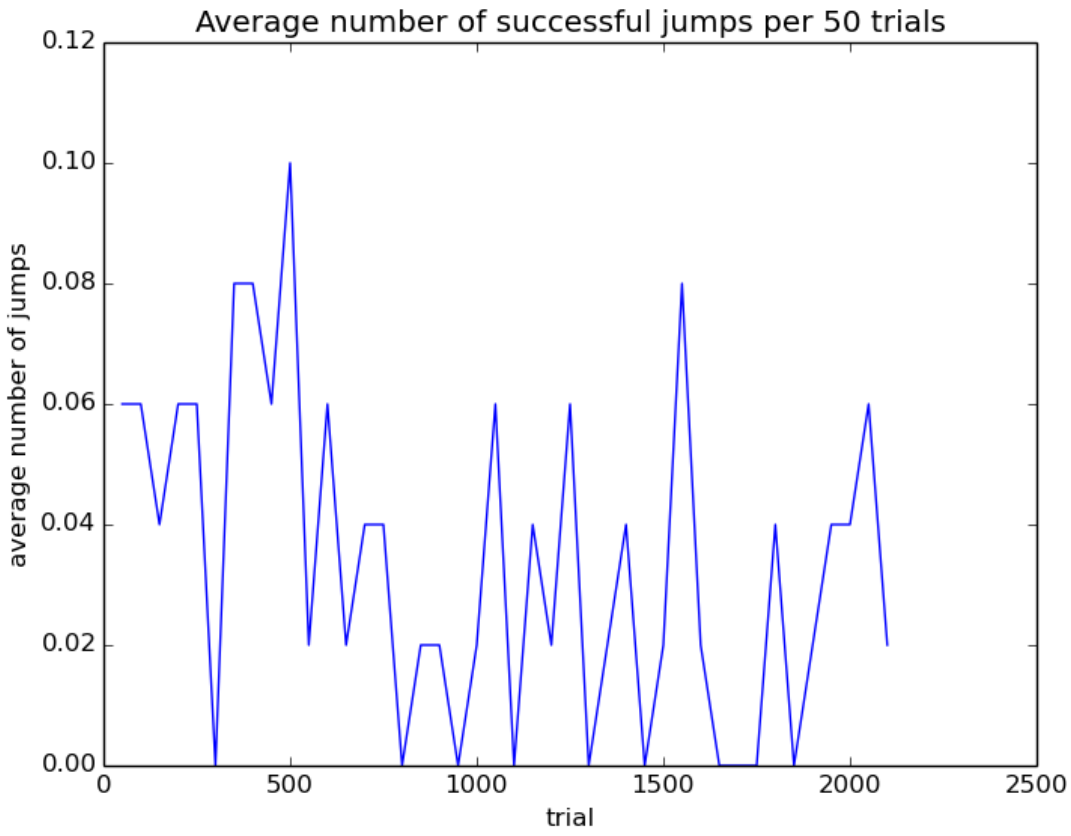

Figure 2: Average number of successful jumps with $\alpha = 0.005$, jump penalty $= -10$, idle reward $= 3$, success reward $= 50$, and dying penalty $= -25$

Our final two experiments were with randomized values for our features, in an attempt to examine the effect of various changes to parameters.

Experiment three had the following settings:

$$\begin{aligned} \alpha &= 0.8682662725534351 \\ \text{jump penalty} &= -7 \\ \text{idle reward} &= 4 \\ \text{success reward} &= 11 \\ \text{die penalty} &= -16 \end{aligned}$$

The results of this experiment are displayed in Figure 3.

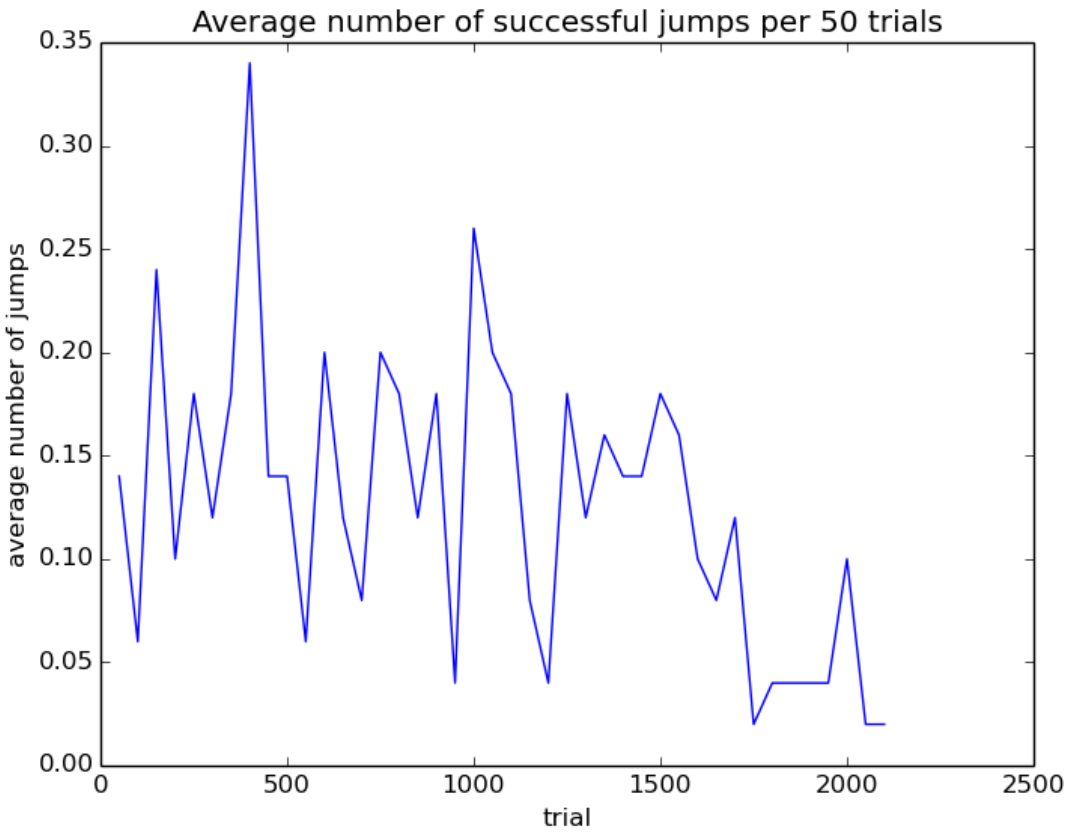

Figure 3: Average number of successful jumps with $\alpha = 0.8682662725534351$, jump penalty $= -7$, idle reward $= 4$, success reward $= 11$, and dying penalty $= -16$

Experiment four had the following settings:

$$\begin{aligned} \alpha &= 0.6564771414639463 \\ \text{jump penalty} &= -4 \\ \text{idle reward} &= 4 \\ \text{success reward} &= 34 \\ \text{die penalty} &= -18 \end{aligned}$$

The results of this experiment are displayed in Figure 4.

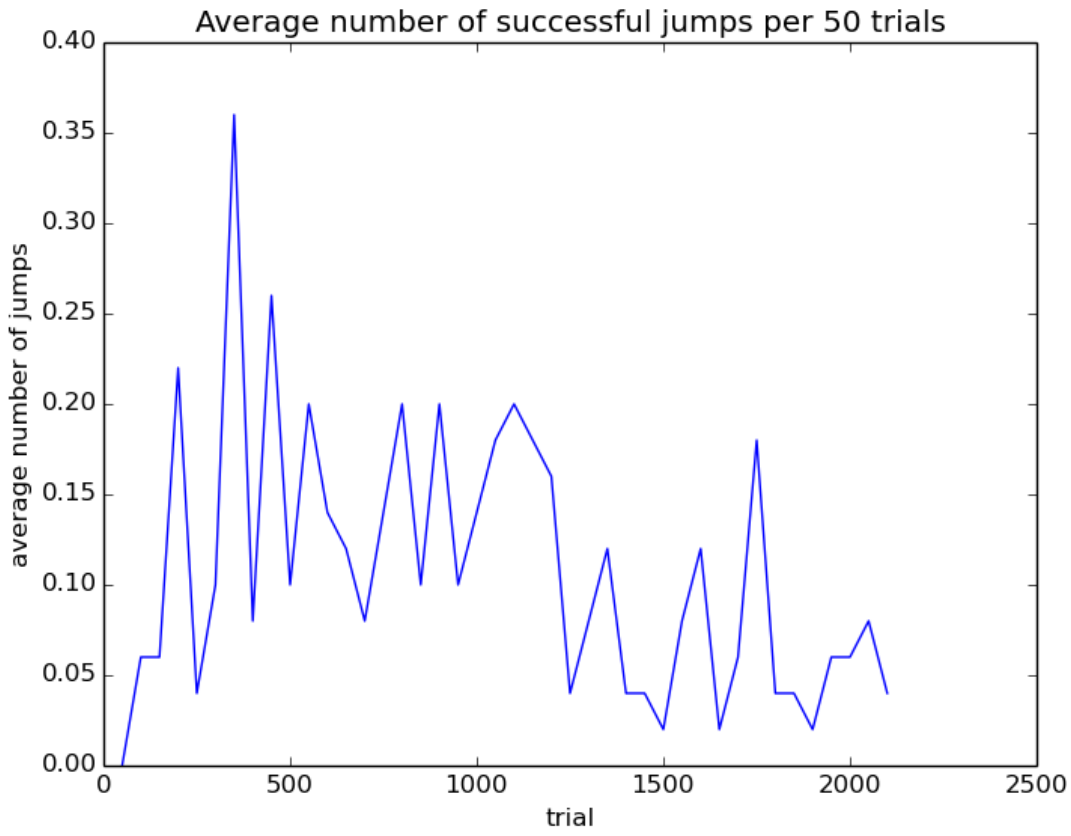

Figure 4: Average number of successful jumps with $\alpha = 0.6564771414639463$, jump penalty $= -4$, idle reward $= 4$, success reward $= 34$, and dying penalty $= -18$

4.3 Discussion

As can be seen from our graphs, none of our experiments were able to correctly "solve" the T-Rex game. While our first experiment was able to average around 3 successful jumps per trial future experiments were more chaotic and did not perform as well.

After experiment one we noticed that the agent had simply decided that the best course of action was to simply jump constantly. For our next experiment we attempted to curb this behavior by imposing a jump penalty, less than the death penalty, hoping the agent would still want to jump to avoid death but not jump needlessly. This was not the case, it opted to simply never jump.

Our future experiments were done to examine the effects of varying our parameters in various ranges, though largely unsuccessful.

We feel that our chosen features (speed, y-position of the T-Rex, x-distance to the nearest obstacle, y-position of the nearest obstacle) are well-chosen, and do correctly capture the important information for an agent to determine whether to jump or not.

However the notion of rewards/penalties is more complicated. After our first trial we noticed that the T-Rex was constantly jumping, and we decided ourselves that this was "undesired" behavior. However the agent decided this due to our assignment of rewards/penalties. During our second experiment with imposing a "jump penalty" the agent decided that jumping at all wasn't worth it, and simply collected the "idle reward" until it died on the first obstacle.

While we feel confident our features are important information for the agent to know, the rewards and penalties clearly have a large effect on the agent's behavior. There are many options

here, from reward/penalty values, to choosing different behaviors to reward/penalize.

5 Conclusion

Ultimately, our goal of teaching an agent to play the T-Rex game was only partly successful, in that we were able to teach the agent to gradually perform better, but only in a very limited fashion. Additionally, the learning process was quite slow, so that the learning that did happen took quite some time, though this was not entirely unexpected.

Something that would improve on this project in the future would be to explore the parameter space much more thoroughly, thus allowing us to find a combination of parameters that would optimize the agent's learning. We could also incorporate the concept of rewarding or penalizing the action of ducking.

One of the largest issues we ran into with this project was deciding how to reward/penalize the agent. Our goal was, essentially, to have the agent successfully run this obstacle course for as long as possible. So it might seem natural to reward the agent for being alive for various amounts of time. However, if the agent never lives to receive that award, it doesn't know to try and reach it.

Effectively the agent, after "finishing" the learning period, can only emulate the best run that it has experienced. If the agent never stumbles upon a good assignment of parameters that lead to a good run, then it is essentially crippled into doing poorly always.

One way to combat this, would be to increase the starting value of ϵ (our experiments started with a value of 0.20). This effectively has the agent start with trying to emulate its best run, rather than randomly searching. Admittedly this does cripple our experiment, but even with this small starting value, our experiments take around two hours to reach a "learned" state.

References

[1] Andrej Karpathy. REINFORCEjs. `http://cs.stanford.edu/people/karpathy/reinforcejs/`.

[2] Carlos Henrique Costa Ribeiro. *A Tutorial on Reinforcement Learning Techniques*. Technological Institute of Aeronautics São José dos Campos, Brazil.

[3] Andrej Karpathy. Deep reinforcement learning: Pong from pixels. `http://karpathy.github.io/2016/05/31/rl/`, May 2016.

[4] PyBrain - The Python Machine Learning Library. `http://pybrain.org/docs/index.html`.

[5] Sarvagya Vaish. Flappy Bird RL. `http://sarvagyavaish.github.io/FlappyBirdRL/`.

[6] Simon's Techincal Blog - PyBrain: Reinforcement Learning, a Tutorial. `http://simontechblog.blogspot.com/2010/08/pybrain-reinforcement-learning-tutorial.html`.