A Comparative Study of Deep RL on Chrome's T-rex and Snake Games

Rahul Chunduru 160050072
Suraj Narra 160050087
Sai Charith 160050083
Saiteja Talluri 160050098
{chrahul,surajnarra,charith,saitejat}@cse.iitb.ac.in

Abstract

In this project, we implement and analyze the effectiveness of various end-to-end deep reinforcement learning-based strategies to play video games, in particular, the classic Snake game and the less-studied JavaScript-based Chrome offline T-rex game. We employ CNN networks to model the policy and Q-functions. Finally, we conclude by comparing the performance metrics of these algorithms with suitable optimized approaches and heuristic approaches and present a detailed analysis of the favourable conditions for different algorithms based on the game.

1 Introduction

Although Reinforcement learning has been around for several decades now, due to the advent of Neural Networks, there has been a renewed interest in the field. The development of super human competent agents such as AlphaGo and Google's DeepMind are testament to this. These agents are so well designed and trained that they exceed grand master level of play many times over and their abilities are humanly incomprehensible. However at the heart of their designs lies the fundamental RL-algorithms such as Q-leariníng and Policy gradient algorithms. These algorithms have been thought to be infeasible due to the huge state space and computations required for their application in human cognitive games. However, Neural Nets and computation intensive hardware architectures now made this possible.

Given such a game environment E, scoring system P and set of actions A, the problem statement for the project is to compute efficient RL-based strategies mapping from game state to actions i.e., $T : S \rightarrow A$, to score high following P and analyze their performance across different games.

So for this project, we consider the popular Snakes game and the relatively less studied Chrome's T-rex game for the analysis.

2 Our Approach

We use already existing Javascript implementation for the T-rex game and implement a bidirectional communication channel between the browser/javascript code and the python AI program to perform actions (jump and duck) and extract the game state. For the snake game we have implemented the game in Python. Once the game is running, we capture screenshots from the game and perform image pre-processing to extract the required features.

2.1 Modeling the Game as MDP

To capture the state of the game S, we consider single game state as a group of sampled screenshots in short intervals of the game play. The set A will be bit-vectors encoding each of the possible action that the agent can take. In particular, '*no action*' is also part of A which is the default action taken at all times, when user doesn't press any buttons.

2.2 Deep RL algorithms

We are going to explore the variants of the following Deep Reinforcement Learning algorithms in the context of Snake and T-rex game. In both these algorithms environment is captured from a sequence of frames up to the current time at fixed intervals.

- Deep Q-learning Network(DQN)
- Policy Gradient

2.2.1 DQN

In this method, we would like to mimic Q-learning using a Neural Network which approximates the Q-function. It takes the current state

(S) and action (A) as inputs and outputs the corresponding Q-value. Based on the reward obtained at time-step t, the Q-value of the state (s^t) and taken action (a^t) can be updated as,

$$err_t \leftarrow r^t + \gamma \times max_a Q_t(s^{t+1}, a) - Q_t(s^t, a^t)$$

$$Q_{t+1}(s^t, a^t) \leftarrow Q_t(s^t, a^t) + \alpha_t err_t$$

where γ is Discount factor, r^t is the reward obtained by taking action a^t in state s^t and α_t is the learning rate. Analogously, in DQN, to achieve the update a loss function like

$$loss = err_t^2$$

is back-propagated through the network to train the weights.

2.2.2 Policy Gradient

In this method a Neural Network is used as a proxy for the optimal policy. The neural network takes a sequence of frames up to current instance as input and gives a distribution over actions. All the actions which led to the termination of the game are penalized by back-propagating a gradient of -1. We may incorporate small positive reward in case of some favorable actions like Snakes consuming the food. In this way the network figures out which actions lead to longer survival time and score.

3 Implementation

3.1 Snake Game

3.1.1 Environment

At any moment one pixel is food, randomly generated each time it is eaten. Standard rules of Snake apply - hitting the wall or its own body kills the snake, consuming food increases its body length by one. The state is one frame - 3 channels (RGB) of 121 pixels each. There are three possible actions - turn left, turn right, keep straight. We reward the agent only for consuming food, and penalize the end of a session. No image processing is required for Snake, as there are fewer pixels than T-Rex and none of them can be ignored.

3.1.2 Baseline

We implemented a heuristic based on graph search. At each step, the snake will avoid actions that will crash into its body or the boundary, and try to avoid moving away from it body, thus preventing loops with empty space in them. To have a sense of direction, the snake will take action that will reduce the distance to the food, ties are broken at random. This method is surprisingly effective, resulting in maximum lengths of around 40 in a 9×9 board. We have been unable to even come close to this number with Reinforcement Learning methods, so our baseline would be random selection.

3.2 Chrome T-rex Game

Chrome T-rex game is the Google's offline game consisting of a T-rex striving to dodge obstacles, such as cactus and birds, and surviving as long as possible. The T-rex is able to perform three actions: “jumping”, “ducking” and “going forward”.

3.2.1 Environment

We used the T-rex browser game extracted from the javascript files and made available open source by Wayou Liu. Our agent directly interacts with the game running in the browser and we did not use an emulator, thus preventing any slow down in the frame rate of the game. We extracted the game state from the browser and implemented a duplex channel to allow for communication between our agent and the browser. This channel is necessary to pass information such as actions and game states.

3.2.2 Preprocessing

We didn't directly use the images we received from the javascript game as states, we preprocessed them before using them as the inputs to our DQN and A2C algorithms. This accelerates the training as we eliminate noisy parts from the image. We apply the following preprocessing steps on this image.

- Extracted a region-of-interest by removing the score portion and other unnecessary information to right side in the frame.
- Remove harmless objects such as clouds by filtering them out based on the colours i.e., lighter colour objects are harmless in the game, so we filtered them out.
- Standard step (based on Atari games) of resizing the image to 80 x 80 pixels.

Finally we stack multiple images (to be precise last 4 frames) into a single state which serves as a kind of memory. Thus we boiled down the initial gray scale frame of 600x150 to finally 4x80x80.

3.2.3 Learning

We applied Deep Q-Learning and Actor Critic algorithms for the T-Rex game and reported the average and maximum score in the results section. The base code for the algorithms has been taken from the OpenAI gym and modified according to our game making suitable changes to interact with the JavaScript browser.

3.3 Algorithm Implementation

3.3.1 A2C Learning

We implemented $A2C$ (Actor-Critic learning) using feed-forward and CNN architectures. Here, the networks takes in the state as input and output the Softmax distribution over the actions along with estimated state value of the next state (critic). The agent then samples as per the distribution and interacts with the environment (Actor). The updates to the network are done using in batches after generating sufficient episodes.

3.3.2 CNN-DQN Learning

We implemented ϵ_t-greedy Q-learning using CNNs. Here, the network takes in state s as input and gives the Q values over actions. The agent chooses greedy action or an exploration with ϵ_t probability. Once a sufficient length of episode is generated, the loss function is computed as the squared difference of target Q-value and current Q-value (err_t).

Some latest improvements in the DQN network we used are described below.

Experience Replay
Instead of using only a single SARSA observation in the error function $\mathcal{L}(\theta)$ we used a batch of observations. This breaks correlations in the data, reduces the variance of the gradient estimator, and allows the network to learn from a more varied array of past experiences. We stored the SARSA observations in a FIFO queue, referred to as memory and then sampled from the memory to get new batch of experiences.

Target Network
In order to avoid oscillations while training , we used a different network, called the target network, to estimate the Q-values during several epochs. The target network has fixed parameters, which reduces the variance and makes the learning more stable. We update the target network parameters with the values of our main network periodically. To be more precise the loss will be

$$\mathcal{L}(\theta) = (\mathcal{Q}_\theta(s,a) - (r + \gamma \times max_{a'} \mathcal{Q}_{\theta_*}(s', a'))^2$$

where $\mathcal{Q}_\theta(.,.)$ is our main network and $\mathcal{Q}_{\theta_*}(.,.)$ is our target network. Both networks have the same architecture but can have different values for the parameters which are periodically updated from main to target network.

Duelling structure to the CNN architecture
The idea behind a dueling structure is to decompose the Q value into two parts. The first is the value function V(s) which indicates the value of being in a certain state. The second is the advantage function A(a) which tells how much better taking a certain action would be compared to the others. We can then think of

$$Q(s,a) = V(s) + A(a)$$

The goal of Dueling DQN is to have a network that separately computes the advantage and value functions, and combines them back into a single Q-function only at the final layer. We had two fully connected layers at the end of our model which compute the value and advantage function. We then average the output of these two parts to end up with the final Q value.

4 Experiment Details

For both the games, to speed up training, we ran multiple instances of the agent in parallel. We trained each model for 1e7 steps, updating weights in batches of 512 steps over all instances. The loss function is minimised using ”Adam” Optimiser and updates are stochastic. The feed-forward models took 30 minutes to converge, convolutional architectures around 2 hours for each of a2c and DQN.

5 Results and Analysis

We present the performance of different agents for each game in the Tables 1 & 2.

From the tables, it can be seen that, $A2C$ algorithm performs better and also converges faster for both games. However, $A2C$ scores also have more variance.

Agent / Game	A2C	CNN-DQN	Heuristic	Random
Snake	11.6	6.2	34.3	3
T-Rex	70	1200	NA	40

Table 1: Average score of the games as played by different agents.

Agent / Game	A2C	CNN-DQN	Heuristic	Random
Snake	15	8	40	4
T-Rex	130	1600	NA	70

Table 2: Maximum score of the games as played by different agents.

6 Conclusion

In this work, we looked into image based-deep RL agents for the Snake and T-rex game. In particular we looked the two popular Deep-RL algorithms - DQN A2C. While A2C converges faster, it has more variance as it converges to local optima more often. In contrast, DQN is more computational expensive.

References

Niels Justesen, Philip Bontrager, Julian Togelius, Sebastian Risi. 2019. Deep Learning for Video Game Playing.

Zhepei Wei, Di Wang, Ming Zhang, Ah-Hwee Tan, Chunyan Miao, and You Zhou. Autonomous agents in snake game via deep reinforcement learning. In *2018 IEEE International Conference on Agents (ICA).*

Junjie Ke, Yiwei Zhao and Honghao Wei AI For Chrome Offline Dinosaur Game. 2016. *Project Report* for Stanford ML Course.

Vincent Dutordoir. `https://vdutor.github.io/blog/2018/05/07/TF-rex.html` 2018. *Blog on Tensorflow T-rex Implementation*

Oscar Knagg. `https://github.com/oscarknagg/wurm` 2018. *Blog on Torch Snake A2C Implementation*