DinoML: Reinforcement Learning and Neuroevolution on the Chrome Dino Game

Ajinkya Pawar
ajinkya.pawar@iitgn.ac.in
Indian Institute of Technology
Gandhinagar, Gujarat, India

Jitender Kumar
jitender.kumar@iitgn.ac.in
Indian Institute of Technology
Gandhinagar, Gujarat, India

Pranshu Kumar Gond
pranshu.kumar@iitgn.ac.in
Indian Institute of Technology
Gandhinagar, Gujarat, India

Sagar Bisen
sagar.bisen@iitgn.ac.in
Indian Institute of Technology
Gandhinagar, Gujarat, India

ABSTRACT

"OpenAI's Dota 2 AI steamrolls world champion e-sports team with back-to-back victories". This is news from 2019, and it is a big deal because Dota 2 is a fairly complex multiplayer game and the fact that an AI bot was able to beat human world champions at the game using deep reinforcement learning algorithms reflects the power of these algorithms in solving complex problems. In this project, we teach AI based models to learn versions of the famous Dino game which we come across in the Chrome browser when we do not have an internet connection. We use a *Deep Q Network* which is a popular reinforcement learning agent, where the *Q* comes from the Q-Learning algorithm in reinforcement learning and stands for 'quality'. We use a famous tutorial on Deep RL by Ravi Munde, *"Build an AI to play Dino Run"* [7] and modify it to obtain better performance and results. Finally, we also implement the NEAT algorithm to train the Dino game and analyze the performance of this algorithm.

CCS CONCEPTS

• **Computing Methodologies → Machine Learning**.

KEYWORDS

deep reinforcement learning, neural networks, NEAT, genetic algorithm, q-learning

ACM Reference Format:
Ajinkya Pawar, Jitender Kumar, Pranshu Kumar Gond, and Sagar Bisen. 2021. DinoML: Reinforcement Learning and Neuroevolution on the Chrome Dino Game. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

1.1 DinoGame

Dino Run game, is a single player game. The task for the player is to control a constantly running T-Rex dinosaur to make it dodge obstacles in its path. At any point in the game, the T-Rex can perform one of the three actions which are *jump, dodge* and *do nothing*, while continuously running. As the game progresses, the speed of the T-Rex increases. The player keeps getting points for as long as he/she is able to save the T-Rex from hitting any obstacle, but when this happens, the game ends and the current score is the player's score for that game.

1.2 Reinforcement Learning

Reinforcement Learning is a separate branch of Machine Learning. The central idea of RL models is to try to solve a problem by trial and error while constantly learning from a feedback mechanism. RL involves terminologies such as *agent*, *environment*, *actions*, *state* and *reward* [5]. With respect to a video game, an agent is the AI algorithm that learns to play the game. So, the algorithm is the player. The environment is an entity which, basically can tell what happens to the game if the agent takes a particular action. A state of the environment is a condition that the game can be in at any time step. In simpler terms, Reinforcement Learning is about the agent interacting with the various states of the environment by taking actions that it can perform, in a state, going into the next state, and learning from it's experience. The goal of the agent is to master the game, i.e. to maximize its total score, which is more formally called the *cumulative reward*. A primary algorithm to achieve this is the Q-learning algorithm, in which an entity called a Q-table is learnt [2]. The rows correspond to states and columns correspond to actions. An entry of the table is called the *Q-value* for that state-action pair (s, a), and quantifies how good it is for the agent to take the action a when in state s.

1.3 Deep Q-Learning

In a dynamic environment, a state can be a collection of consecutive frames of the game, but it's not a discrete quantity. So in order to use Q-learning in video games, we use Deep Reinforcement Learning, where instead of the Q-table, a neural network is learnt. This neural network learns to predict the Q-values for all actions from a state. So in a nutshell, deep reinforcement learning involves a neural

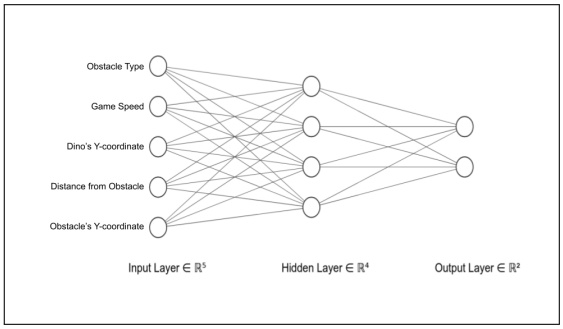

Figure 1: The Feed Forward Neural Network of a genome in the population

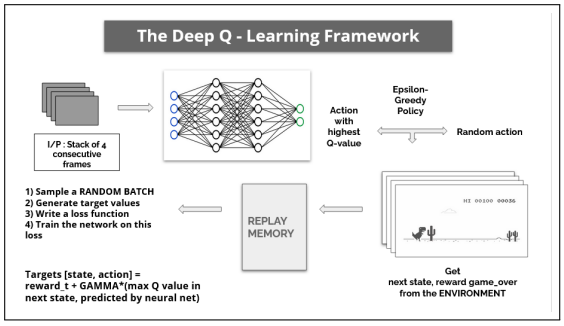

Figure 2: The Deep Q-learning Framework

network learning to predict Q-values with Q-learning being the underlying algorithm for training.

1.4 NEAT

NEAT or Neural Evolution of Augmenting Topologies is an evolutionary algorithm used to create Artificial Neural Networks (ANN) [10]. Taking inspiration from the biological process of evolution, the NEAT algorithm is initialised with a set of genomes (*Population*). Each genome has two sets of genes, i) Nodes: the neurons in the ANN ii) Connection: specifying a single connection (or edge) between the neurons. A *fitness function* is maintained to assign fitness to each genome in a population. The algorithm iterates over multiple *generations* striving to find the fittest genome by making the genome structure more and more complex. This is achieved by reproduction (*crossover*) and *mutation* phases where two topologies are chosen to create a new topology (crossover) and from this newly created topology, mutated sets of other topologies are formed (Mutation).

2 RELATED WORK

Model-based Reinforcement Learning is quite efficient in learning complexities within a system. However they are not as efficient and only provide an approximate optimal solution [8]. A rather more effective algorithm that is used in video games is the traditional Q-learning Algorithm [4]. Deep Reinforcement Learning builds upon the traditional Q-Learning algorithm and uses Deep Neural Networks to predict the Q-value of a state-action pair [6]. The Double Deep Q-Learning algorithm (Double DQN) [2] has the loss function defined in such a manner that it minimizes the upward bias generated in traditional Q-Learning algorithms [4].

NEAT has been used in playing many video games like *flappy birds* [1] and video games under the Atari framework [3]. It has also been used to training an agent to play *Go* [11]. rtNEAT is a variation of the NEAT algorithm where evolution takes place in a real time environment (Like NERO) unlike the tradition method of iterating over multiple generations [9]. cgNEAT or Content Generating NEAT is used to generate custom content in a video game based on the user preferences [3].

3 METHOD

3.1 Deep Q-Network : Model Architecture

3.1.1 Framework. The following control flow was implemented in the training phase (Figure 2) . It starts with an input to the neural network, which is a set of 4(hyper-parameter) consecutive frames of the game. The neural network then makes a prediction for the Q-values for each action possible from this state. Based on an epsilon-greedy policy, either a random action is taken else the one with the highest Q-value is taken. This experience is stored in the replay memory. After regular time intervals, a batch is sampled from the replay memory. An error function is computed using the Q-learning algorithm and the neural network is trained on this error.

3.1.2 Our Contribution.

(1) Environment API - We implemented a custom environment class that would serve as the intermediary for communication between the agent and the game. Both, the neural network and the game are in a single python program. In the implementation that we referred to, *selenium web driver* is used as the medium of communication between the game running in the browser. It requires *openCV* libraries to take screenshots of the screen and then generating input for the neural network, which is computationally expensive [7]. When the game is in pygame, no screenshots are required as the frame of the game in each time step is present in a specific object, maintained by pygame.
(2) STATE - We also redefined a state based on our understanding of a state. In our logic, we assume that a snapshot of the game with the dino in air is not a state, because a new action while already in air is not possible, and hence should not be fed to the neural network. So if the dino takes a jump action in some state, then the next state will be the game snapshot after the jump is completed stacked 4 times (Figure 3) .

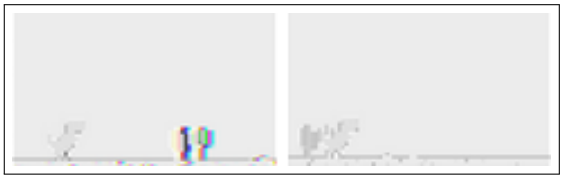

Figure 3: Game STATES before a jump action and after a jump action respectively

3.2 NEAT : Model Architecture

As mentioned in section 1.4, a population of genomes is maintained in each generation. Each genome was described by the feed-forward neural network shown in Figure 1.

The input layer of the neural network was fed with the following set of features i) The type of obstacle Dino encounters (Cactus or Pteranodon) ii) The current game speed iii) The Y-coordinate of Dino's image (top) in the screen iv) The Euclidean distance between Dino's image and the obstacle's image v) The Y-coordinate of obstacles' image (top) in the screen A hidden layer with 4 nodes was added. The output layer consisted of 2 nodes. The activation function at each layer was set to be the sigmoid function. Hence, the output nodes gave the confidence upon which Dino's next move was made. The population in each generation was set to be 150. The *elitism* or the number of genomes guaranteed to be in the next generation was set to 2. Only 20 per cent of the population was allowed to reproduce each generation (*survival threshold*). The fitness of a genome was described by the points the genome was able to score in the game.

4 EXPERIMENTAL SETTINGS

4.1 Deep Q Network

We trained three neural network agents with different weights initialization (Figure 5), on 2 variations of the dino game, variation based on game parameters (Figure 4). While training a model-game pair, we store the evolution of game score with time and the loss with time respectively.

Version	Gravity	Game_speed	Jump_speed
1 (easy)	0.6	4	11.5
2 (difficult)	1.2	8	12

Figure 4: Gravity and JumpSpeed decide the projectile motion of the dino. GameSpeed decides the amount of movement of obstacles towards the dino in a single frame. Therefore, increasing this game parameter makes the game faster. Version 2 will have a jump twice as small as in version 1 and a game speed double the game speed in version 1. Hence we consider 1 to be easy and 2 to be difficult.

4).

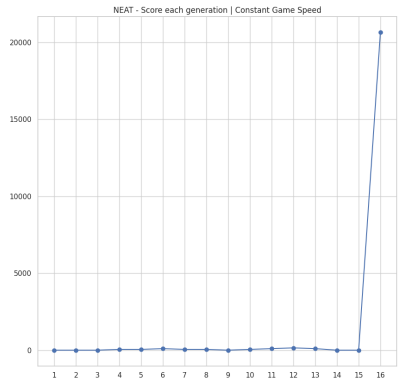

Figure 6: Score per generation (Constant game speed; X-axis - Generation, Y-axis - Score)

Model	Game	Weights Initialization
Model 1	Version 1	Random
Model 2	Version 2	Trained weights from Model 1
Model 3	Version 2	Random

Figure 5: Specifics of models trained

4.2 NEAT Algorithm

We used the *pygame* library in Python to implement the Dino game and the *python-neat* library to evaluate the performance of the NEAT algorithm.

We used two setups to evaluate the performance of the NEAT Algorithm.

i) Evaluating the NEAT algorithm on a game with constant game speed i.e. the speed with which objects move in the screen (Same as Version 2 of the game in section 4.1).

ii) Evaluating the NEAT algorithm on a game with increasing game speed.

5 RESULTS AND DISCUSSION

5.1 Deep Reinforcement Learning

Below are the inferences we can make from the evaluation of Deep Q Network algorithm on the Dino game,

(1) For each model, as the agent plays more and more games, the score increases (Figure 11) , and the loss decreases, converging to 0 eventually (Figure 10) .
(2) Model 1 and Model 2 take almost the same number of games for learning. This could be because Model 1 was trained on the easy version but with random weights initialization, and Model 1 was trained on the difficult version but with trained weights as initial weights.
(3) Model 3 takes the highest number of games, more that 1.5 times the number of games required by Model 2. A possible

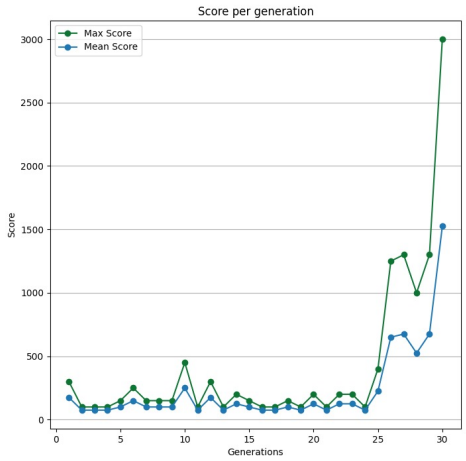

Figure 7: Maximum and Mean Score each generation (increasing game speed)

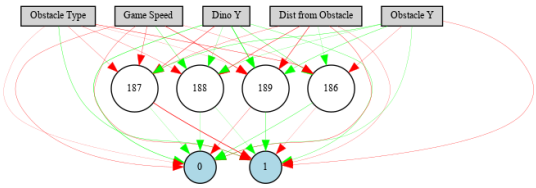

Figure 8: The Winner genome that gives the best score.

reason is that both were trained on the difficult version but the weights initialization again made the difference.

(4) We observed a slight difference in the nature of learning of Model 1 and Model 2. When we put these agents in an even more difficult version of the game, where we made the jump size such that the dino can cross an obstacle only if it jumps very late. Model 2 is still able to play this version because it has learned to jump only when the obstacle is very close. Model 1 however, jumps relatively a little early, so e it cannot cross some obstacles and thus performs poor.

The referred implementation [7] was trained on version 1 of the game. Our Model 1 performs better that this implementation in terms of the the highest score achieved and the approximate number of games required for training (Figure 9) . Also, our model was trained on CPU whereas the baseline model required GPU.

Model \ High Score	200	500	1000	4000	16000
Model 1	500	700	850	850	850
Baseline	2500	3200	4500	>6000	Did not reach

Figure 9: Each cell is the number of games required by our model and baseline model to reach the high score in each column

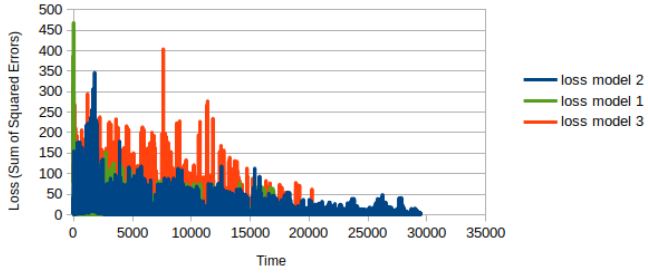

Figure 10: Evolution of loss with time. Loss was computed after every 5 time steps on a random batch of size 16 selected from the Replay Memory

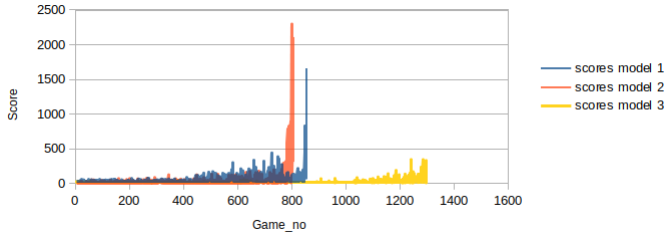

Figure 11: Evolution of game score per game

5.2 NEAT Algorithm

Below are the inferences we can make from the evaluation of NEAT algorithm on the Dino game,

(1) When the game speed is set to be constant, it took almost 16 generations for the genome to learn when to duck and when to jump.

(2) The neural network of the winner genome or the genome that gave the best score is shown in Figure 8. As mentioned in section 1.4, the connections and the nodes together constitute the gene of a given genome. Thus, after performing the NEAT algorithm, connections (or edges) are built. The weights assigned to these edges are also determined by the algorithm itself.

(3) The red arrow in Figure 8 correspond to a positive correlation while the green arrow correspond to a negative correlation. Hence, by looking at the neural network structure of the winner genome, inferences can be made about which features contribute the most to dino's jump move or duck move.

6 CONCLUSION

The Deep Q-learning approach fetches us satisfying results. However the NEAT algorithm with it's ease of implementation and the time it takes to create the winning ANN is worth acknowledging. Both the approaches have their pros and cons. The Deep Q-learning approach is harder to implement and it staggers in a game with increasing game speed due to the high computational load. With NEAT, due to the random nature of the game, it is not always guaranteed to get a player that can achieve the desired fitness.

REFERENCES

[1] Matheus G. Cordeiro, Paulo Bruno S. Serafim, Yuri Lenon B. Nogueira, Creto A. Vidal, and Joaquim B. Cavalcante Neto. 2019. A Minimal Training Strategy to Play Flappy Bird Indefinitely with NEAT. In *2019 18th Brazilian Symposium on Computer Games and Digital Entertainment (SBGames)*. 21–28. https://doi.org/10.1109/SBGames.2019.00014

[2] Hado van Hasselt, Arthur Guez, and David Silver. 2016. Deep Reinforcement Learning with Double Q-Learning. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence* (Phoenix, Arizona) *(AAAI'16)*. AAAI Press, 2094–2100.

[3] Erin Jonathan Hastings, Ratan K. Guha, and Kenneth O. Stanley. 2009. Automatic Content Generation in the Galactic Arms Race Video Game. *IEEE Transactions on Computational Intelligence and AI in Games* 1, 4 (2009), 245–263. https://doi.org/10.1109/TCIAIG.2009.2038365

[4] Beakcheol Jang, Myeonghwi Kim, Gaspard Harerimana, and Jong Wook Kim. 2019. Q-Learning Algorithms: A Comprehensive Classification and Applications. *IEEE Access* 7 (2019), 133653–133667. https://doi.org/10.1109/ACCESS.2019.2941229

[5] Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore. 1996. Reinforcement Learning: A Survey. *CoRR* cs.AI/9605103 (1996). http://dblp.uni-trier.de/db/journals/corr/corr9605.html#cs-AI-9605103

[6] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei Rusu, Joel Veness, Marc Bellemare, Alex Graves, Martin Riedmiller, Andreas Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. 2015. Human-level control through deep reinforcement learning. *Nature* 518 (02 2015), 529–33. https://doi.org/10.1038/nature14236

[7] Ravi Munde. 2020. Build an AI to play Dino Run. https://blog.paperspace.com/dino-run/

[8] Vitchyr Pong, Shixiang Gu, Murtaza Dalal, and Sergey Levine. 2018. Temporal Difference Models: Model-Free Deep RL for Model-Based Control. *CoRR* abs/1802.09081 (2018). arXiv:1802.09081 http://arxiv.org/abs/1802.09081

[9] Kenneth Stanley, Bobby Bryant, and Risto Miikkulainen. 2006. Real-Time Neuroevolution in the NERO Video Game. *Evolutionary Computation, IEEE Transactions on* 9 (01 2006), 653 – 668. https://doi.org/10.1109/TEVC.2005.856210

[10] Kenneth O. Stanley and Risto Miikkulainen. 2002. Evolving Neural Networks Through Augmenting Topologies. *Evolutionary Computation* 10, 2 (2002), 99–127. http://nn.cs.utexas.edu/?stanley:ec02

[11] Kenneth O. Stanley and Risto Miikkulainen. 2004. Evolving a Roving Eye for Go. In *Genetic and Evolutionary Computation – GECCO 2004*, Kalyanmoy Deb (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1226–1238.