Playing Dino Run with Deep Reinforcement Learning

Ravi Munde
Northeastern University
ravi72munde@gmail.com

Abstract

In this project, a deep learning model to learn optimal action patterns from visual input using reinforcement learning is implemented. The model can play Dino Run, the Chromium browser's offline game, using Q-learning by approximating the future expected reward for possible next actions. The model is trained on processed game frames and predicts the outcome of next possible actions. The feature learning is done with the help of 3 convolution layers and a Deep Q-Network optimizes the policy of actions. The model parameters are optimized for faster CPU training without losing important game features. The model can detect obstacles and help the Dino agent avoid them by learning the timing of different actions.

1. Introduction

Learning to control an agent's action in an environment has been traditionally looked at as a supervised learning problem. A model can be easily trained by observing expert players' gameplay recordings. The environment and the actions to which the model is exposed solely depends on the human's decisions. Even though we can reach a good accuracy with this approach, the chances of the model performing better than humans are rare. Moreover, the agent might miss out on actions that can perform better but were not explored as experts tend to exploit the actions which give a good reward. Reinforcement learning can help the model to self-learn by observing the environments by performing certain actions and the reward these actions result into. The reinforcement learning aims to maximize the rewards given any set of actions bring when performed by an agent in a state of the environment.

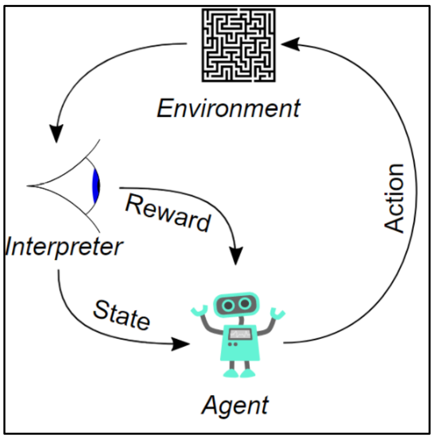


Figure 1 Reinforcement Learning framework by Megajuice - Own work, CC0, https://commons.wikimedia.org/w/index.php?curid=57895741

The paper uses a model-free Q-learning algorithm to find an optimal solution through a Markov Decision Process. A vanilla deep learning model faces many challenges while implementing a Q-learning algorithm including but not limited to sparse and noisy data, delayed rewards, correlation between states, credit assignment problem and huge time for training [1]. This paper implements a variant of Q-learning algorithm with experience replay mechanism where we train model on random batches of previous transitions to alleviate the above-mentioned problems and smoothen the training distribution. Additionally, Convolutional Layers were incorporate to handle the problem of sparse data and improve the speed of learning.

2. Background

Dino Run is an endless runner where the agent, Dino, performs actions to survive as long as possible by either jumping over or ducking at obstacles. The environment variables like obstacle types and speed vary as the game progresses. As the training is targeted on CPU system, this variation is minimized for faster convergence. The environment is restricted to a constant speed and limited to a single type of obstacle. The agent, Dino, interacts with an environment in a sequence of actions, states and rewards i.e. it takes an action a_o in state S_o, transitions to state S_1 and receives a reward r_1 for the transition. This single transition, represented as (S_o, a_o, r_1) is referred as an experience of the agent. When in S_1 state, it repeats the sequence with action a_1. This sequence of transitions can be represented as ($S_o, a_o, r_1, S_1, a_1, r_2, S_2, a_2, r_3, S_3, a_3, r_4 \ldots\ldots\ldots\ldots S_n$) and holds the experience history and helps make up a Markov Decision process. Agent's aim is to maximize the future reward but our environment is stochastic so the discounted future reward is calculated. It can be represented as

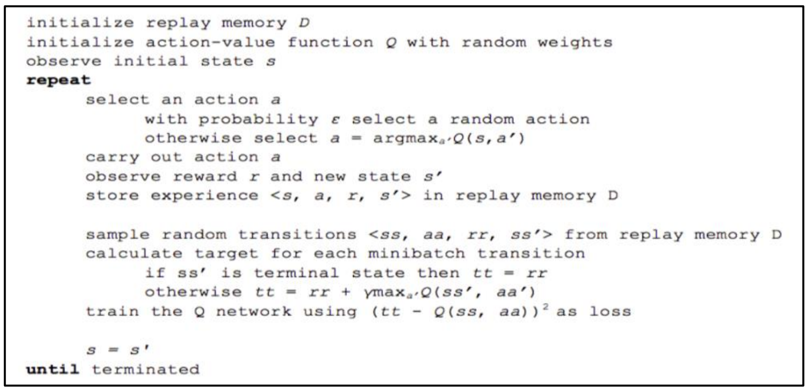
$$\boldsymbol{R_t = r_t + \gamma r_{t1} + \gamma^2 r_{t2} \ldots + \gamma^{n-t} r_n = r_t + \gamma R_{t1+1}}$$

where γ is the discount factor between 0 and 1 and measures how much effect does the past actions have on future rewards. In case of Dino, the agent cannot perform any action while it is in mid-air due to the previous jump, so $\gamma = 0.9$ is chosen.

In Q-learning, we define a function $\boldsymbol{Q(S_t, a_t) = max\ R_{t+1}}$ representing maximum discounted future reward. When we have this Q-function for all possible action we choose the action with maximum Q value

$$\boldsymbol{\Pi(S) = argmax_a Q(S, a)}$$

where π is the policy to choose an action in each state. We can iterative approximate the Q-function and using a deep neural network



```
initialize replay memory $D$
initialize action-value function $Q$ with random weights
observe initial state $s$
repeat
    select an action $a$
        with probability $\varepsilon$ select a random action
        otherwise select $a = \text{argmax}_{a'} Q(s, a')$
    carry out action $a$
    observe reward $r$ and new state $s'$
    store experience $<s, a, r, s'>$ in replay memory D

    sample random transitions $<ss, aa, rr, ss'>$ from replay memory D
    calculate target for each minibatch transition
        if ss' is terminal state then $tt = rr$
        otherwise $tt = rr + \gamma \text{max}_{a'} Q(ss', aa')$
    train the Q network using $(tt - Q(ss, aa))^2$ as loss

    $s = s'$
until terminated
```

Figure 2 Deep Q-learning algorithm [6]

To read the pixel data, which represent the states, convolution layers can be added to better detect and learn the features. In case of Dino, the screenshots from the screen are taken and fed to the model. A traditional Supervised Model would see the image and give out probabilities for each action. We choose the one with the highest probability. However, in case of Q-learning, the output of the model is the maximum Q-value for all actions we get through approximating the Q-function. This turns our problem into a regression one rather than classification. The loss function can be given as

$$\boldsymbol{L = \frac{1}{2}[r + max_{a'}Q(S', a') - Q(S, a)\,]^2}$$

Where S' is the state resulting from state action a in state S. a' is the next action

3. Related Work

A 2013 paper by DeepMind on "*Playing Atari with Deep Reinforcement Learning"* was a breakthrough with phenomenon results for playing Atari 2600 games. The model used a similar approach of convolutional neural network and DQNN learning the games. The technique was implemented on seven Atari games where the model performed better than human experts for three of them and outperformed all the available techniques.

Another well-known example of the reinforcement learning is the TD-gammon where the model self-learned the game and achieved a human level gameplay. It used a multi-layer perceptron to approximate the Q-function. TD-gammon was the source of inspiration for the DeepMind's paper.

Deep Reinforcement Learning for Flappy Bird by Kevin Chen uses a similar approach to DeepMind for playing Flappy Bird. This model performed better than human experts.

4. Pre- Processing

The game environment is implemented in Chromium while our RL model is implemented in python. Selenium is used for interfacing between the game environment and the model i.e. to control the Dino agent. PIL & OpenCV captures the screenshots for processing. The original screenshots of the entire game are high dimensional 1200x300 RGB pixels. Modeling this large raw state-space is computationally expensive and time consuming for learning relevant features. To reduce the dimensionality of the images, we modify the images before feeding it to the network as the target training system is a CPU only.

4.1 Game Environment Pre-processing

To play the game efficiently, the model needs to learn the non-pixel features from environment too. These features include the speed of the agent, obstacle types and dimensions, different possible actions, and acceleration. Learning these features would take additional iterations and a longer time for convergence. Limiting the environment variables can speed up the learning process significantly as observed because the agent is exposed to fewer variations. The changes in environment included acceleration set to zero, limiting the type of obstacles to one, only using jump as the possible action, removing game-over screen, removing high score panel. The image below compares the original game and modified game screens.

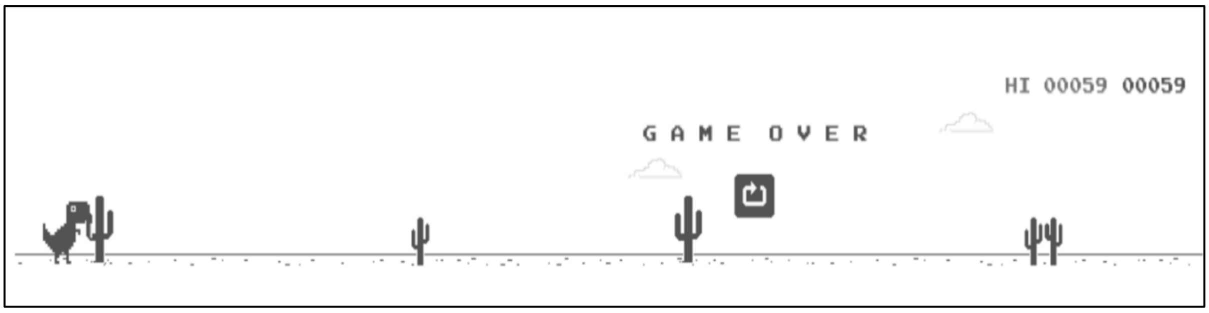


Figure 3 Original game

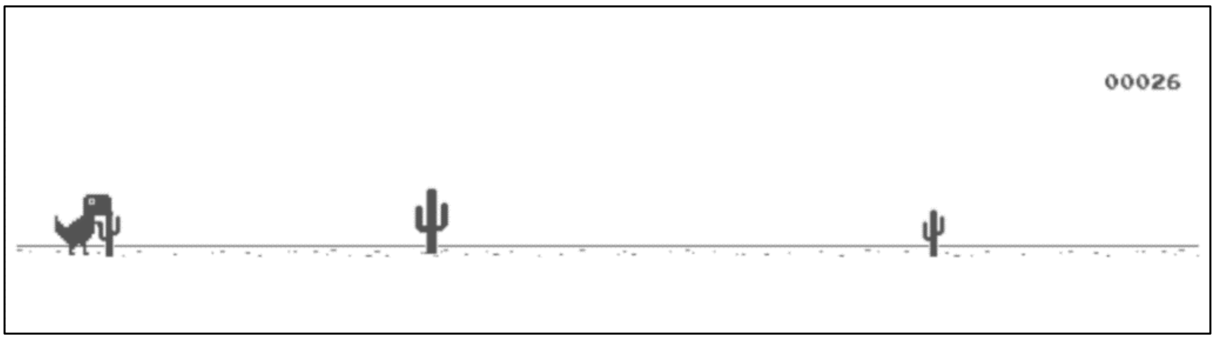


Figure 4 Modified game with removed features

4.2 Pixel Based Pre-processing and Feature Extraction

The original pixel dimensions of the game are 1200 x 300 RGB making a single input of size 1200x300x3. For learning the speed of the agent, we need to stack together 4 images as a single input. Thus, the overall dimension of the input is 1200x300x3x4 which lead to significant frame rate drop while capturing the screenshots and made the game unplayable on a CPU only system. The images are processed to reduce these dimension by using a pipeline of processors in OpenCV. Using selenium, browser size is reduced to 200x300. Region of interest(ROI) is captured using PIL & OpenCV reduced to it a minimal size of 40x20x3. Even though the image had RGB channels the game did not have any colors so greyscale conversion cannot reduce the features, instead, OpenCV's canny filter was used to get rid of the background, the clouds and highlight the edges. Then 4 consecutive frames are stacked together to learn the speed [1]. As seen in the last transition below, the agent is cropped out of ROI. It was observed that the agent does not have any relevant feature while playing the game other than its distance from the obstacle. The edge of the frame can be used for measuring the distance without adding any noise.

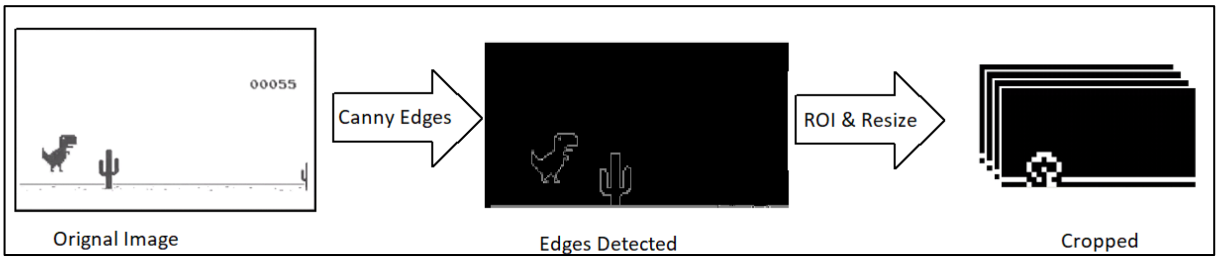
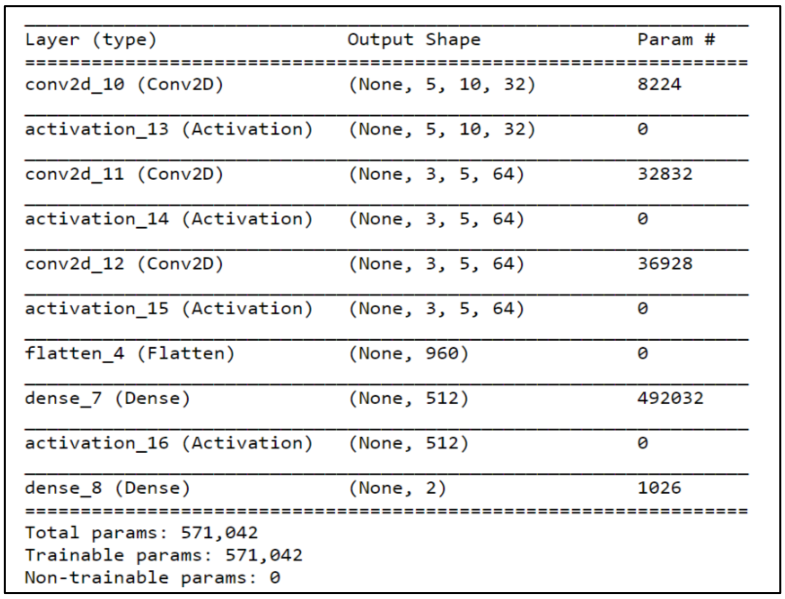


Figure 5 Canny Edges Detection and Region of interest extraction

5. Network Architecture

5.1 Deep Convolutional Network

The Deep Neural Network Model consists of three Convolution layers with two fully connected layers. The table below shows the dimension of each layer. Pooling layers were not utilized as we do not intend the network to be insensitive to the location of an object in the image. In this game, the position of objects matters as we are observing distance from obstacle. The Convolution Layers accept the four-dimensional input of images and flatten the data before passing it to the fully connected layers. Layers are processed with a stride of (4,4), (2,2) & (1,1) respectively. The final output has the dimension that matches the number of possible actions. In this paper, the agent is restricted to only two actions (Jump & Do nothing). The output is the Q-value for each action (0: Do Nothing, 1: Jump) and we choose the action with highest Q-value. The model achieved a good stability at a learning rate of 0.0001.



```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_10 (Conv2D)           (None, 5, 10, 32)         8224
_________________________________________________________________
activation_13 (Activation)   (None, 5, 10, 32)         0
_________________________________________________________________
conv2d_11 (Conv2D)           (None, 3, 5, 64)          32832
_________________________________________________________________
activation_14 (Activation)   (None, 3, 5, 64)          0
_________________________________________________________________
conv2d_12 (Conv2D)           (None, 3, 5, 64)          36928
_________________________________________________________________
activation_15 (Activation)   (None, 3, 5, 64)          0
_________________________________________________________________
flatten_4 (Flatten)          (None, 960)               0
_________________________________________________________________
dense_7 (Dense)              (None, 512)               492032
_________________________________________________________________
activation_16 (Activation)   (None, 512)               0
_________________________________________________________________
dense_8 (Dense)              (None, 2)                 1026
=================================================================
Total params: 571,042
Trainable params: 571,042
Non-trainable params: 0
```

Figure 6 Model Summary with trainable parameters

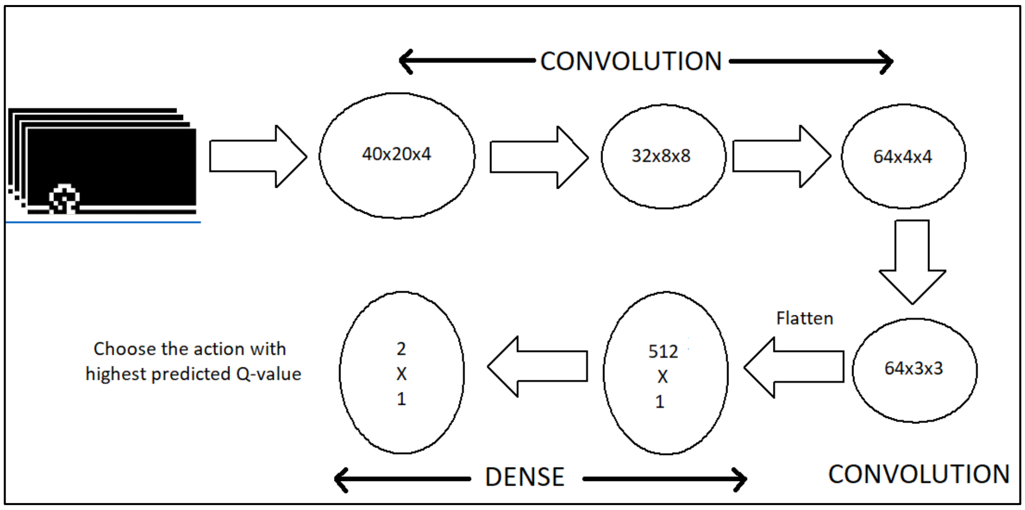


Figure 7 Network Architecture with layer details.

5.2 Deep Q-learning Parameters

In addition to the network hyperparameters, the Q-learning process requires parameters that control the model learning:

- **Gamma/Discount factor (γ) :** This decay rate decides the degree to which the model sees in the future while performing an action. Value can range between 0-1, 0 making the model short sighted and 1 deterministic [6]. For the Dino Run model, 0.99 gave best results.

- **Observations:** The training is done in batches instead of continuous for stability and faster processing of gameplay. We observe the gameplay before we start training making sure that we have enough data to start back propagation. The model observed the gameplay with random actions for 50,000 timesteps before making predictions for the actions

- **Replay Memory:** The training is performed on random batches extracted from the replay memory. *Replay memory* decides the number of previous steps we consider while training [6]. In this paper, the replay memory size is 50,000 steps

- **Exploration Factor:** To alleviate the problem of exploitation of same actions in supervised learning, reinforcement learning uses the principal of random exploration [1]. The model performs a random action for certain number of steps instead of predicting the action. *Explore* parameter decides the number of steps the model can take random actions. For this paper, the exploration factor is set to 100000. The reason for high exploration factor is the tendency of the agent to continuously jump instead of detecting and jumping obstacles.

- **Epsilon / Randomness factor:** As we explore additional actions, we need the randomness to decrease over time. Higher epsilon value corresponds to more randomness in actions while learning [8]. The model starts with an epsilon of 0.1 and continues to decay till it reaches 0.0001 over the course of 10000 exploration steps. Initially the model starts with decent randomness and reduce the randomness of actions as model starts to learn.

- **Reward:** The *reward* is the outcome which model receives after each action. A positive reward indicates a good action and a crash gets negative reward [6]. For the Dino model, the reward was calculated dynamically to also help learn game progress. This was done using the fact that the game score increases with distance traveled. Following reward calculation mechanism was utilized. A relatively higher reward is given to ‘do nothing’ to introduce a bias to avoid jump only strategy.

  Survived with ‘do nothing’: 0.01xScore

  Survived with ‘jump’: 0.009*Score

  Crashed: -11/score (11 is the minimum score in any game)

6. Results

The model was trained for approximately 2.6 million steps, each utilizing a single image from the gameplay. However, the initial 1 million steps were used for parameter tuning and observations which produced noisy learning steps. The last one million steps shown below display a steady decline in loss function(MSE) and fluctuates around 0.4. Initial steps had random spikes in error due to parameter modifications. The CPU only system slowed down the stability due to random frame rate drops.

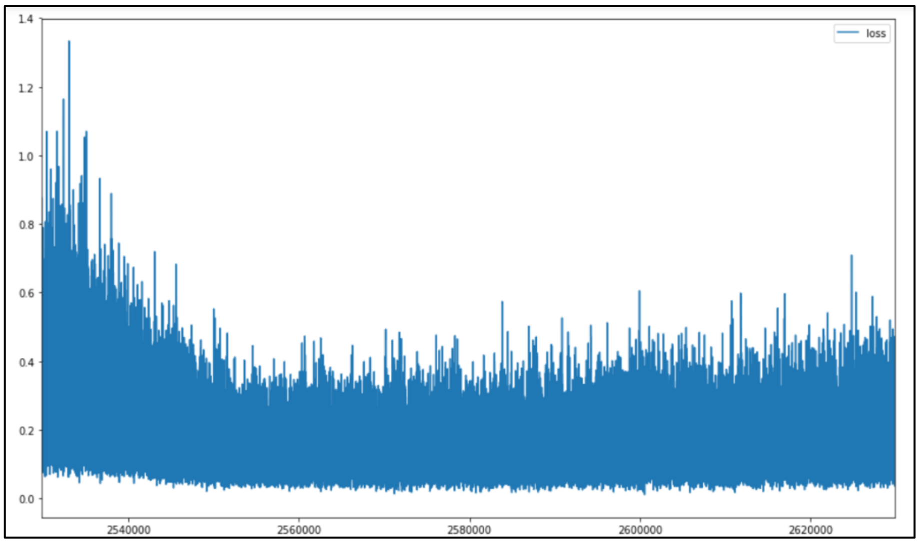


Figure 8 MSE loss function for last million steps

Even though we observe a stabilizing loss function, a better measure of stability would be the game scores achieved at the end of each game. The initial 100,000 gameplays were used for model parameter tuning the highest scores observed were around 60. The drop in the score at 100,000th gameplay is due a major change in q-learning parameters and introduction of the dynamic reward. The model starts to perform well after 100,000 gameplays or the 1 millionth step achieving a maximum score of 265 frequently scoring more than 100.

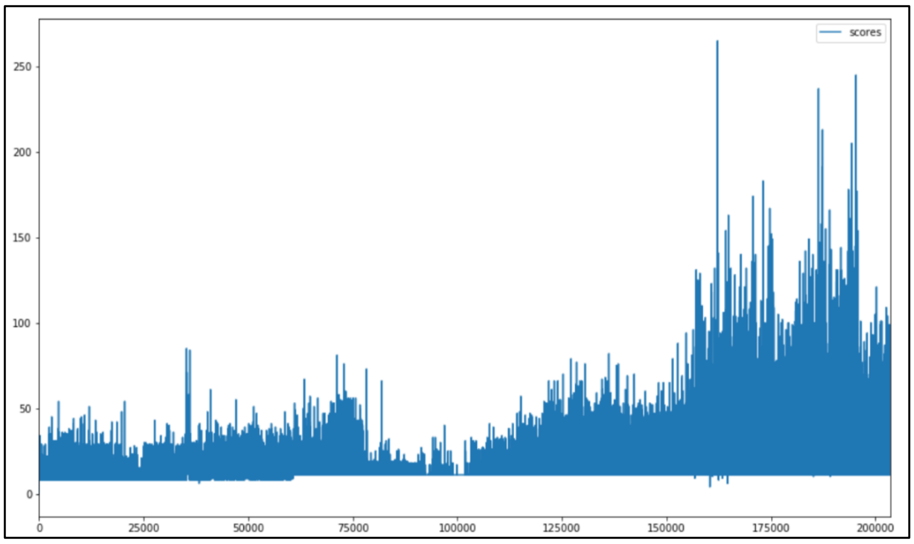


Figure 9. Game scores

7. Current limitations and Future Work

The model starts to perform well as compared to initial steps, however, few limitations inhibit its ability to learn faster and score better. The model does not consistently score high due to random frame-rate drops that occur while learning on a CPU only system. Moreover, the small size of image (40x20) coupled with the current model architecture possibly leads to some loss of features and slower learning. Additionally, the model was trained on a controlled game environment with no variation in speed or obstacles. The real game has additional features which the model was not exposed to for comparatively faster convergence. The model also learned that higher jumping rates would bring higher rewards which can be observed by the distribution of actions below. Ideal gameplay has a very low density for jumps. A human's game play was record for the score of 500 for comparison. The figure shows the distribution of actions of AI and Human.

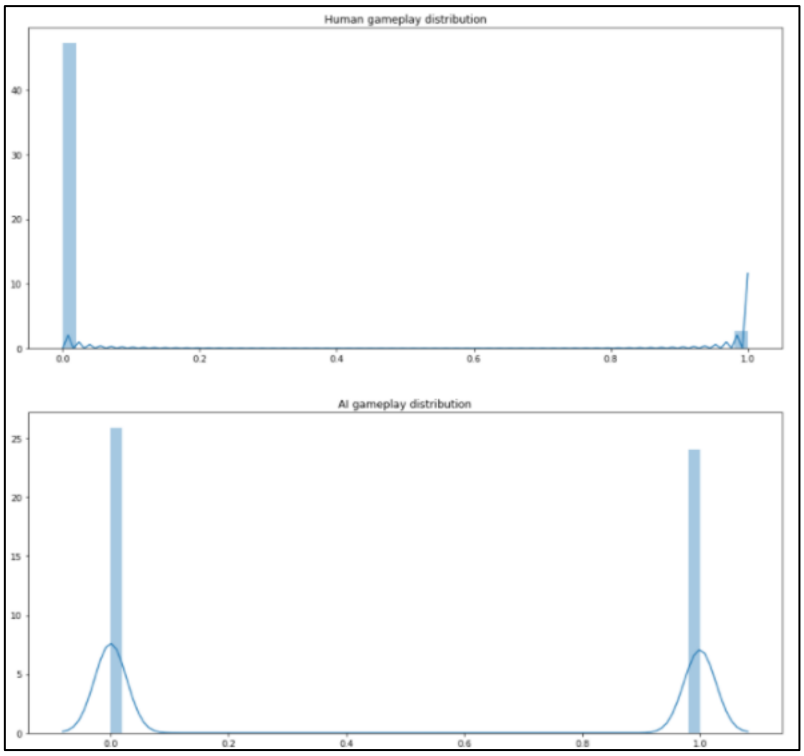


Figure 10 Human vs AI Density Distribution of Actions.

This model, however, set a base for future works where this learning can be transferred to other models minimizing the initial learning noises in an unfamiliar environment. A GPU based training will certainly give better and faster learning because constant and good frame rate [8]. The model can then be trained in phases where it is exposed to more variations in environment at each phase. The model should be able to handle speed variations and more types of obstacles. Another phase would be to allow an additional action of ducking at flying obstacles. A bigger challenge would be to use this model and apply to a different game environment with similar actions. This technique should give the model higher initial performance as compared to a fresh model.

8. Conclusion

I studied an implementation of Reinforcement Learning using a Deep Convolution Neural Network for controlling actions of an agent to survive in an endless game environment. A CPU system was targeting for training. Game features were reduced by pre-processing the environment as well as inputs. This reduction in features certainly made the gameplay and parallel training possible but did not improve the learning rate significantly. The model was however able to detect an obstacle and act on it. The minibatch and experience replay approach for training, introduced by the DeepMind paper, was also validated. The model provides a solid foundation for future trainings and shows promising improvements with each training step.

References

[1] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou,

Daan Wierstra, and Martin Riedmiller. *'Playing Atari with Deep Reinforcement Learning'* arXiv:1312.5602, 19 Dec 2013

[2] Kevin Chen, *Deep Reinforcement Learning for Flappy Bird.*

[3] Gerald Tesauro. *Temporal difference learning and td-gammon*. Communications of the ACM, 38(3):58–68, 1995.

[4] *Toy example of a deep reinforcement learning model playing a game of catching fruit*, https://github.com/bitwise-ben/Fruit

[5] MNIH, Volodymyr, et al. *Human-level control through deep reinforcement learning*. Nature, 2015, vol. 518, no 7540, p. 529-533.

[6] Tambet Matiisen. *Demystifying Deep Reinforcement Learning* https://ai.intel.com/demystifying-deep-reinforcement-learning/

[7] Sascha Lange, Thomas Gabel, and Martin Riedmiller. *Batch Reinforcement Learning*

[8] *Using Deep Q-Network to Learn How To Play Flappy Bird* https://github.com/yenchenlin/DeepLearningFlappyBird