Artificial Intelligence for T-Rex Running

Zhengrong Wang, Bisheng Huang, Kaiwen Huang, Aozhu Chen, Ning Xin
Department of Computer Science
University of California, Los Angeles
{seanzw, bhuang16, kaiwen1, achen28, nxin}@ucla.edu

***Abstract*—In this project, we present an artificial agent to play the game T-Rex Running based on reinforcement learning. We applied both Q-Learning and Deep-Q-Learning to train our agents. We incorporated additional variables and features in the learning process to deal with high dimensional input and varying game velocities. We deployed our implementation with the game extracted from Chromium and customized the user interface with selectable models to evaluate our agent's performance in scoring. The best scores our agent can achieve is much higher than that of a human player. And our results show a significant improvement in score with more features added in the model; meanwhile some features such as velocity has more importance in affecting the final score than others.**

Keywords—Reinforcement learning, Q-learning, Artificial Intelligence

I. Introduction

Artificial intelligence has been widely applied to game playing over the past years. Computer-trained agent can compete with human professional players such as the success of AlphaGo developed by Google's DeepMind. Reinforcement learning is a common framework used in games where actions are taken to maximize the reward. In this project we developed an artificial intelligent agent for the famous Chrome game T-Rex Running.

Traditional table-based Q-Learning requires to quantify the state space. However, this is not possible for those games with high dimentional state space. The challenges lying here consists of two parts: we have to derive efficient representations of the environment and to generalize the model into new situations. Deep-Q-Learning is a framework that solves both problems. It combines Q learning, a reinforcement learning model, with deep neural network; it has been proved to achieve performance comparable to a human player in a diverse range of Atari games[1]. In this project, we aim to develop an agent the Chrome offline game called T-Rex Runner with both Q-Learning and Deep-Q-Learning and compare their result. We will also investigate the significance of different features in the agent's performance by training our agent with different features.

II. T-Rex Running Game

The offline dinosaur in Chrome is a game called T-Rex Running. Google Chrome users are probably familiar with the T-Rex dinosaur that shows up when your computer is not connected to the Internet. T-rex dinosaur had short arms and therefore lot of things were out of its reach. Chrome, like that dinosaur, too is having trouble reaching the Internet. The error page has featured a tiny, 8-bit T-Rex, but you can hit the space bar to start a runner-style mini game. The T-Rex will run and your mission is to prevent it from bumping into obstacles. The game keeps track of your high score, and if you accidentally land on an obstacle you can always retry. Either that, or you can stop playing and try to figure out why your internet isn't working.

A. World

Your job is to press the spacebar to make the T-Rex jump across cactus and pterodactyl. During the game the T-Rex will gradually accelerate from 6 to 13 to incress the difficulty. Fig 1 illustrates the coordinate of the world we will be using. The x axis is horizontal and y axis is vertical. Fig 2 shows all the possible obstacles. Notice that cactus has different size and pterodactyl has different hight and speed.

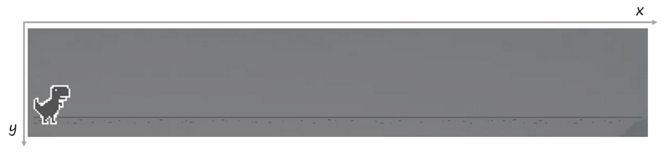

Fig. 1: Coordinate of TRex World

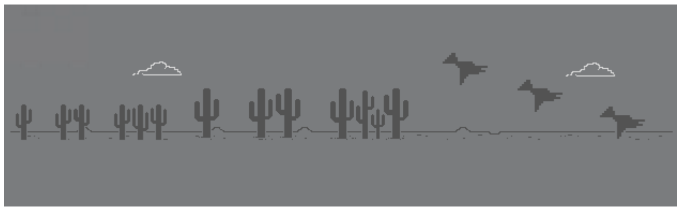

Fig. 2: Different Obstacles In the Game

B. Challenges

1) Speed of bird can have a positive or negative speed offset.
2) Speed is gradually increasing.
    - Speed should be crafted into the states
3) Obstacles have different characteristics
    - Bird and cactus
    - Positions, width and height are random
4) Actions are not continuous
    - Once it is in the air, it cannot jump again
    - Different from other games, e.g. flappy bird

5) Computational power is limited
    - Training was performed in the browser
    - Hard to achieve millions of iterations even with speed up mode

III. Background knowledge

Before we going to Q-Learning, we need to know the basic idea about learning. Learning is essential for unknown environments, when designer lacks omniscience. Secondly, learning is useful as a system construction method, expose the agent to reality rather than trying to write it down. Lastly, learning modifies the agent's decision mechanisms to improve performance. It is difficult to hand-design behaviours that act optimally (or even close to it). Agents can optimize themselves using reinforcement learning which means not learning new concepts (or behaviours), but given a set of states and actions it can find best policy. Learning can be done online and continuously throughout lifetime of agent, adapting to changing situations. A learning agent can be thought of as containing a performance element that decides what actions to take and a learning element that modifies the performance element so that it makes better decisions.

A. Q-Learning

Because the agent only observes the current frame at any one time-step, the task can only be partially observed and many of the states of are perpetually aliased meaning the current screen xt is not enough information to understand the entirety of the agents in-game circumstance. Therefore, we consider a sequence of actions and states and learn strategies from these and use Q-Learning algorithm to solve these problems. The equation is as following:

Q-Learning:

$$Q^*(s,a) = \max_{\pi} E[r_t + \gamma r_{t+2} + \gamma^2 r_{t+2} + ...|s_t = s, a_t = a, \pi] \tag{1}$$

Approximate

$$Q^*(s,a) = E[r + \gamma max_{a'} Q^*(s',a')|s,a] \tag{2}$$

Note here $\gamma \in$ [0, 1] is the discount factor that determines the trade-off between short and long-term rewards. The Q-Learning function tries to learn a score for each legal action under state s with maximum future rewards. The optimal action-value function obeys an important identity known as the Bellman equation. It is the basic idea of many reinforcement learning to use Bellman equation as iterative update. Normally, we would use a table to store the state-action value and update it directly. However, for many games, the state space is too large and its impossible to directly store all states in the memory as it requires quantification. Thats where neural network comes in, which is used to approximate the action-value function.

However, directly applying standard online Q-Learning with neural network has mainly two problems. The first one is there is strong correlation between samples. Suppose now the agent tends to move to the left, then most of samples generated will be from the left. Therefore, the agents policy determines the distribution of the samples. Secondly, unlike traditional supervised learning where samples already have a label or target value, here the target value is also computed by the nerual network. It tries to learning something, but the target is also defined by itself. Both problems lead to unstable, even diverged training results.

B. Deep Q-Learning

The solution for the problems in Q learning is as following [1]. For the correlation between samples, we use experience replay. Its pretty simple by maintaining a memory buffer D with size N. The buffer will store the agents previous experience, which is a tuple containing the state s at time t, the action it took, the reward and the next state. For training, we randomly sample a mini-batch from the butter to train.

There are advantages of this approach. First of all, it reuses samples, so it makes the model more efficient to train. Also, since the training batch is randomly sampled from previous experience, it breaks the correlation between samples. Finally, it averages the behavior distribution over many previous states.

For the second problem, we can see that the CNN is used to calculate the target value. The solution is to use a previous network, instead of the current updating one, to compute the target value. At every iteration, we update network Q, which is used to calculate the action value. At every C steps, we update Q-hat with Q. This makes the model more stable to train than standard online Q-Learning. T

IV. Perception

Since we want to the TRex to learn to jump across the obstacles, we need to define what variables it can perceive during the game. Most of the features we defined in this section is related to TRex itself and only the first obstacle, as our little poor TRex cannot see through it.

A. X Position

The most important feature is the x position of the first obstacle. It measures the distance from the first obstacle to the TRex. With this feature, TRex should be able to learn the most basic strategy of jumping when the obstacle is approaching. Fig. 3 shows an illustration of x.

B. Width & Height

However, as mentioned above, the width and the height of obstacles varies. This makes the jumping position for different obstacles maybe different. Therefore, our TRex is capable of detecting the size of the first obstacle, including the width and the height. Fig. 3 shows an illustration of w and h.

C. Relative Velocity

Another thing that changes during the game is the running speed of the TRex. It gradually speeds up during the game and may need different strategy during high speed than low speed. It is reasonable to assume that the TRex can feel its own velocity to the ground. However, as mentioned above, the speed of pterodactyls can have an offset to the ground, if TRex can only feel its own absolute velocity to the ground, it cannot capture the information that a pterodactyl is approaching faster or slower than other normal cactus. Therefore, the TRex should be aware of the relative speed between itself and the first obstacle. Fig. 3 shows an illustration of the relative velocity v.

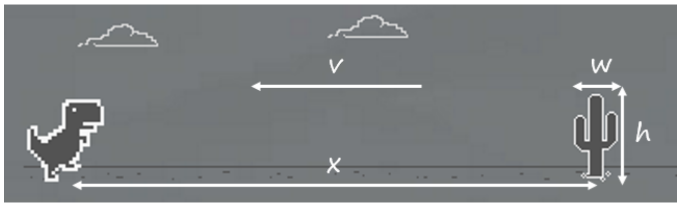

Fig. 3: TRex's perception

V. Model

A. Q-Learning

In this section, we will describe the process of Q-learning in our model.

1) Bucketing the perceptions to features: First, we need to encode the environment, i.e. perceptions of the TRex, into several discrete states. Since the variables in the perception are floating point numbers, we use a certain number of buckets to discretize them. The bucketing process for each variable can be described as followed.

For the X position of the obstacle,

$$\hat{x} = \frac{x}{CANVAS\_WIDTH} * BUCKET_X \tag{3}$$

For the height of the obstacle,

$$\hat{h} = \frac{h}{CANVAS\_HEIGHT} * BUCKET_H \tag{4}$$

For the relative velocity,

$$\hat{v} = \frac{v - MIN\_V + offset}{MAX\_V - MIN\_V + 1} * BUCKET_V \tag{5}$$

$CANVAS\_WIDTH$ and $CANVAS\_HEIGHT$ represent the width and the height of the canvas, respectively. BUCKET denotes the bucket size for that variable. Empirically, $BUCKET_X$ = 20, $BUCKET_H$ = 10, $BUCKET_V$ = 10. MAX_V and MIN_V denote the maximal velocity and minimal velocity of the horizon, and their values are 13 and 6 respectively. $Offset$ is a variable denoting the speed offset of the obstacle. For an obstacle belong to the type of cactus, $offset$ = 0; for a pterodactyls-type obstacle, $offset$ can be either 0.8 or -0.8 with equal probability.

Notice that bucket size is a factor worth considering. A large bucket size will result in a larger state space, which will take more iterations to train, while a small bucket size will not be sufficient to represent the environment unambiguously.

2) Testing the efficiency of the perceptions: In order to learn the amount of information needed to solve the game, we gradually add features to expand the state spaces. The combinations of features we tested in our experiment are listed as followed. The number is the bracket represents the number of all the possible states. Notice that there is an empty state representing no obstacle at sight.

1) X Position
 $(BUCKET_X + 1)$
2) X Position + Height
 $(BUCKET_X * BUCKET_H + 1)$
3) X Position + Height + Velocity
 $(BUCKET_X * BUCKET_H * BUCKET_V + 1)$

3) Ignoring some states when updating the Q-table: In the framework of Q-learning, the agent will constantly choose an action for each state. But in the TRex game, once the Trex jumps into the air, it cannot perform any other actions until it lands on the horizon. To simplify the Q-learning process in our model, Q-table will not be updated during this period. In other words, when the agent chooses the action of jumping, the next state will be the state when the TRex lands on the horizon or hits an obstacle, instead of a state in which TRex is jumping in the air.

4) Reward: For a state in which the TRex collided with an obstacle, the reward was set to -1000. When the TRex is running on the horizon, the reward is 0. When the TRex jumps over an obstacle and lands on the horizon, the reward is 1. The default behavior for the Q-learning agent is doing nothing, which means that if two actions have the same reward for the current state, the agent will choose to stay on the horizon instead of jumping.

B. Deep-Q-Learning

We also tried Deep-Q-Learning algorithm. We use a forward neural network to approximate the Q function.

1) Normalize Features: First, we use the same equation as Eq. 3, Eq. 4 and Eq. 5 to normalize the data. Notice that for neural network the input can be a continuous value. Therefore, we do not have to bucketize the features to discrete values, and the value for each feature is within the range $[0, 1]$. For example, the X position of the first obstacle is normalized with

$$\hat{x} = \frac{x}{CANVAS\_WIDTH} \tag{6}$$

2) Training Detail: As mentioned above, the most challenging part for Deep-Q-Learning is that the network may not converge. Here we implement the algorithm from [1]. There are a few details we want to mention for training.

First, the agent of Deep-Q-Learning does not make a decision for each frame. Instead, it only decides whether to jump or not every 6 frames. This is to make sure that neighboring samples used for training differ from each other and make the training process more stable.

Another thing to notice is that as mentioned in the algorithm, we use an experience buffer and a separate network to compute the target value for the update. The target neural network is updated every 1000 iterations.

TABLE I: API of Agent

API	Arguments	Returns	Description
act	Engine, Reward	Jump or Not	Make the decision, may train the agent.
save	/	Model	Return a serialized model to save.
load	Model	/	Load a model.

3) Network Structure: We use a simple 4 layer neural network. The first layer is the input layer. The second and third layer are both fully connected layer and each contains 64 neurons. Finally, there is a regression layer which outputs the estimated Q value for each state action tuple.

Notice that we have tried more complex network structure and input, e.g. CNN for pixel input, however, the whole experiment is carried out in the browser and the computation power is very limited. It takes more than 2 seconds for the browser to finish one back propagation and we have to focus on handcrafted feature with simple neural network structure rather than raw pixels.

4) Reward: We use a different reward than that for Q-Learning. If the TRex hits an obstacle and died, we give the agent -1 reward. If the TRex passes through an obstacle, either by jumping or doing nothing as the pterodactyl is too high, we give the agent 1 reward. Otherwise, the reward is 0.

VI. Implementation

In this section, we talk more about our implementation.

A. Framework

We the code extracted from Chromium [2] as our basic framework. Then we modify the code so that when updating each frame, an agent can make a decision of whether jumping or not.

We introduce an interface for the communication between the engine and the agent as listed in Table I. The most important API is act(), which takes in the engine instance and the reward of the previous decision. The agent will extract the current state from the engine and make a decision of whether jumping or not. If necessary, e.g. the agent is still learning how to play, it can use the reward and its decision history to train itself.

B. Agents

Given the API, we implement four agents listed as in Table II. Notice that the Human agent will always do nothing so that it will not interfere the human player. The Cheat agent employs a handcrafted AI which uses the information of speed, X position, and height of the first obstacle. It is used along with human player as the baseline.

TABLE II: Agent Implemented

Agent	Description
Human	Always do nothing so no interference to human player.
Cheat	A carefully handcrafted AI.
Q-Table	Use Q Table to learn the game.
Deep-Q-Net	Use neural network to learn the game.

C. Front End Utility

Fig. 4 shows our front-end design and it supports the following functionality.

1) Select Speed
The user can change the simulation speed of the engine. Currently, the engine supports two level of speed: crazy and sober. In sober mode, frames are rendered at 60 FPS, while in crazy mode the engine renders as fast as possible. Notice that this is not the running speed of the TRex and the crazy mode should only be used for training an agent.
2) Change Agent
The user can select different agents from the menu. For some agents, e.g. Q-Table and Deep-Q-Net agent, there are multiple models can be used. Different models utilize different features like the relative speed of the TRex and so on.
3) Save & Load Model
After a long time of training, the user can also save the trained model. It is also possible to load a pre-trained model and let the agent play directly.
4) Save Scores
The user can also save the history scores of the current agent into a csv file. It is useful to illustrate how the agent improves its skill as training goes on.

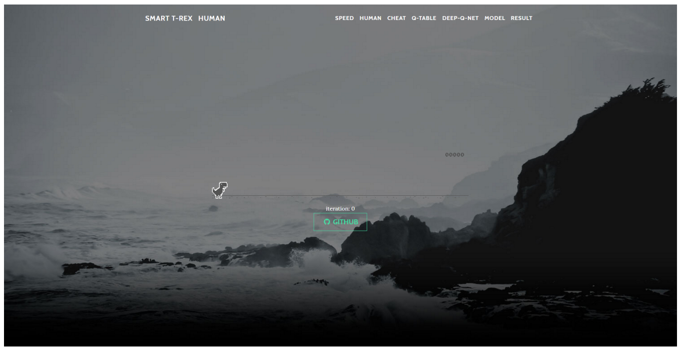

Fig. 4: The Front End Design

VII. Result

A. Q-Learning

1) Experiment Setup: In our experiment, each agent first went through a training process. The number of iterations in the training process for each combination of features was chosen as 100000 empirically, during which the Q-table was updated after each iteration. After training, the performances of each agent were evaluated.

2) Training process: During the training process, whenever the game was over, a game score would be recorded at the corresponding iteration. The relationship between the number of iterations and scores achieved was demonstrated in Fig. 5 and Fig. 6, which corresponded to two set of features, respectively. Note that it will take more than one iteration for the game to end and generate a score. The better the agent performed, the more iterations were needed to generate a score.

It can be observed in Fig. 5 that with only the position of the first obstacle, the game score fluctuated frequently during training. It once got as high as 3000 but later stabilized at around several hundred. This is a typical pattern when the information presented in the feature was not sufficient to solve the game. The key information that was missing in the feature was the velocity of the obstacle. Two scenarios in the game can have the same states but the velocity is different, which will confuse the agent. When the state spaces are small, there is no such solution or strategy that can pass all scenarios of obstacles. In this case, the strategy for the agent might not converge and the performance will fluctuate frequently.

A relatively more robust learning curve can be observed in Fig. 6. With the information of relative velocity of the obstacle, the agents can learn how to act differently when the speed is different. At the same time, the score also grew more slowly due to an enlarged states space.

3) Training Pattern: A typical training pattern for Q-learning agents can be described as followed. Initially, due to the default behavior of not jumping, the TRex will directly run into the obstacle. The crash state can be denoted as S_0, and its previous state can be denoted as S_1. When the collision happens, not jumping at S_1 will be given a penalty, so that next time agent will choose to jump at S_1. Now the agent is able to pass one obstacle, but it is jumping very close to obstacle. It will not change its strategy until two consecutive obstacles come up so that the agent finds it impossible to pass the second obstacle. Therefore, both jumping and not jumping at S_1 are also given a penalty. The agent will try to avoid S_1, and choose to jump at a previous state, say S_2, and so on so forth.

The difficult part of the game is when consecutive obstacles come up with a very small gap. Following this pattern, the Q-learning agent was able to learn the ideal strategy for this game: jump as early as possible for each obstacle.

4) Q-learning Model evaluations: As shown in the result Table III, the score for Q-learning agents gradually increase with the introduction of more features.

With only the position of the obstacle, the score achieved is several hundred. After that it might be hard to deal with the increased speed. With the addition information of height, the agent can achieve a score of over one thousand. Relative velocity is a more important feature than height, as the score for the agent 'QL: Position + Velocity' is two times higher than the agent 'QL: Position + Height', which is also expected.

The agent with all three perceptions already mastered the game and was able to achieve a performance that far exceeded human player and even the handcrafted Cheat AI, reaching as high as 590494.

B. Deep-Q-Learning

Table III also shows the scores for Deep-Q-Learning models. As we expected, it is much more difficult to train neural network than traditional Q-Learning. The biggest challenge is that Deep-Q-Learning requires randomly taking actions during

TABLE III: Score achieved for Each Agent

Agent	# Iters	Max	Min	Average
Human	/	4016	613	2031
Keep Jumping	/	21	21	21
Random Action	/	40	21	25
Cheat	/	62207	6058	22364
QL: Position	100000	945	361	580
QL: Position + Height	100000	1860	361	1163
QL: Position + Velocity	100000	8447	471	2075
QL: Position + Height + Velocity	100000	590494	717	159450
DQL: Position	100000	2033	745	1320
DQL: Position + Height	200000	180	32	105
DQL: Position + Velocity	200000	206	53	130
DQL: Position + Height + Velocity	200000	102	22	67

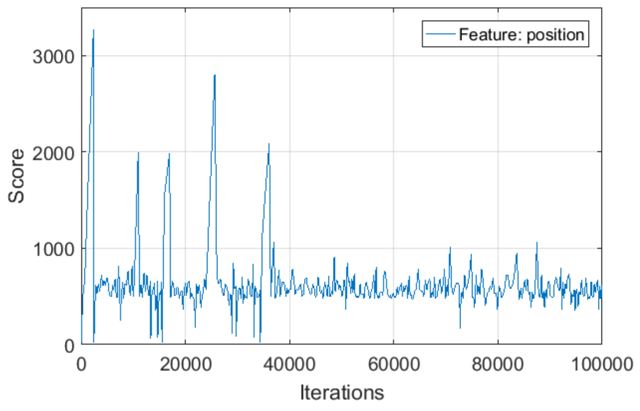

Fig. 5: Training with position

the training phase to explore the state space. However, with random actions, the game usually terminates with a score less than 100, and in this early phase the running speed of the TRex is very slow. As mentioned above, the speed of the TRex is very crucial for the TRex to determine when to jump.

Our experiment shows that with simple X position of the first obstacles, the neural network converges well and Fig. 7 shows the estimated Q value for two legal actions against the X position of the first obstacle. The x axis is the distance to the first obstacle. We can see that the agent is able to learn to not jump when the obstacle is far away (as the Q value of not jumping is higher than jumping) and start jumping when it comes close.

For other Deep-Q-Learning models, additional features do not improve their performance. We think the current network structure is too simple to handle more features. However, the computation of the browser makes it not possible to train more complicate network. For example, we tried the 6 layers' CNN model introduced in [1], and the back propagation for one mini-batch with 32 samples along takes more than 2 seconds. Also, complicate network requires more iteration to train. In [1] they used more than 35 days of gaming experience to train the model, which is impossible for our own PC.

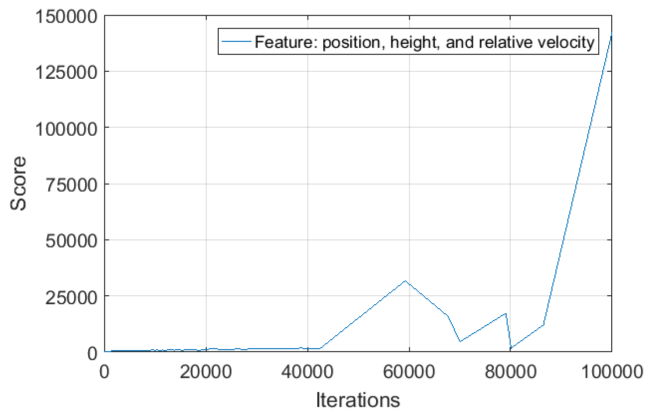

Fig. 6: Training with position, height, and velocity

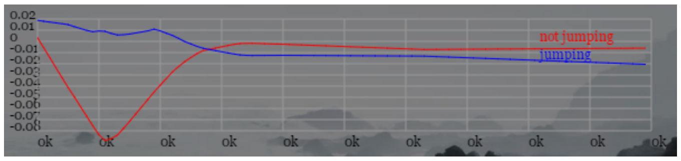

Fig. 7: Q Value of DQL-X position

VIII. Conclusion

As discussed in the training pattern section of Q-learning, the Q-learning agent was able to learn to jump as early as possible for each obstacle, allowing for more space for the next obstacle. With the position, height, and relative velocity of the first obstacle, the agent can learn the essences of the game and was able to achieve a performance that far exceeded human players. It is demonstrated again that reinforcement learning, and more specifically Q-learning, is very promising in training artificial agents in the field of game playing.

As for Deep-Q-Learning, we implemented the algorithm of DQN and trained it with different features. However, due to the limited computation power of the browser, it is not possible to use very complicate neural network, and the simplest model with only the X position of the first obstacle achieves the best result among all the Deep-Q-Learning models we have tried, but is still not as good as models from Q-Learning. We think that a more sophisticated network may be able to utilize additional features and performs better. We can try to export the game to other native platforms, e.g. Python with tensorflow as the backend to enable the usage of more complicated network structure and speed up the training phase.

You can find all the implementation at https://github.com/seanzw/smart-t-rex. There is also a runnable website at https://seanzw.github.io/smart-t-rex/.

References

[1] Volodymyr Mnih et al. In: *Nature* 7540 (), pp. 529–533. ISSN: 00280836.

[2] wayou. *The t-rex Runner Game Extracted from Chromium*. 2017. URL: https://github.com/wayou/t-rex-runner.