Beating the ”Worlds Hardest Game” Using Q-Learning

Shashwat Pratap, Jingkai Sun, Weixi Cai
{shashwat.pratap, w33cai, j283sun}@uwaterloo.ca
University of Waterloo
Waterloo, ON, Canada

Abstract

Obstacle avoidance is an important application of Artificial Intelligence(AI) agents in modern society. In this paper, our objective was to design and train an AI agent to play a popular platforming game: “World’s hardest game”. This game requires players to control a rectangle moving from a starting point to a goal while avoiding obstacles during the process. Inspired by AI agents designed for self-driving, we decided to use reinforcement learning and Q learning to accomplish our objectives. Our final agent can pass the first level of “World’s hardest game” in less than 1000 rounds training. To achieve this, we applied Double-Action Q Learning, which can encapsulate both the movement of the player and the change in environment (due to movement of obstacles) at the same time. We also compared the number of rounds required to beat the level and the survival time of the agent with different learning rates and presented them in line charts.

Introduction

Obstacles avoidance has been a very common topic in platforming video games for years, and recent research in AI video game agents focused a lot on beating obstacles-avoiding games. In this paper, we want to build a AI engine to beat a popular platforming game called “The World’s hardest game”. The premise of the game is deceptively simple but the gameplay is really hard. In the game, the player controls a red block to move from one area to another area, and on the way, the player needs to avoid obstacles and pass some checkpoints while collecting coins.

By beating this game, our AI will be able to perform well in other similar platforming games. Since the game is titled “the world’s hardest game”, it will be cool if we can design and train a general purpose AI to pass the game. This problem is also challenging because we cannot find a relevant paper for this specific game, so we have to read papers about similar games and come up with our own solution.

To build an AI agent to beat “The World’s hardest game”, the most challenging part will be how to dynamically improve the agent’s behaviour. By reading recent research in similar game, we selected Q-Learning model, a reinforcement learning technique to build an AI agent which can perform well in “The World’s hardest game”.

As a result of implementation, our AI agent successfully beats the first level of the Worlds Hardest Game and most of the other human created levels with obstacles with linear movement. We have three main contributions:

- Even though historically, Q learning algorithms have been applied quite a bit to solve platforming games, in our knowledge we were the first ones to apply it on the “Worlds Hardest Game”.
- We implemented the algorithms Q learning and Double Action Q learning on a platforming game with a dynamic environment and compared their performances.
- Double Action Q learning achieves the better performance for the game when compared to Traditional Q learning both in training time and gameplay in human created levels.

Related Work

According to a research at The University of Hong Kong, researchers have compared the double action Q-learning with traditional Q-learning in solving the avoiding obstacle problem like “The World’s Hardest Game”; and found that the double action Q-learning is better on the sum of rewards and steps, with the increase of the number of obstacles. (Yung) (Rahimi)

Another research at Southern Illinois University Carbondale shows how they develop and train dynamic intelligent bots for some basic behaviors using Q-learning. Researchers have used a wide range of algorithms to tackle the problem of collision avoidance. But in the context of obstacle avoidance / platforming games researchers from deepmind have used Deep reinforcement learning to tackle the problem

(Mihai Duguleana)while the researchers at University Transilvania of Brasov used Q-learning along with a neural network planner to tackle the path planning problems (Sallab) Researchers at University of Auckland used modified Sarsa and Q-learning algorithms to micromanage (micro) the players units in the very famous Real Time Strategy (RTS) game - Starcraft Broodwar.(Volodymyr Mnih 2013)

Methodology

An optimal AI agent to beat “The World’s hardest game” would like to find a path to the goal without any collision with the obstacles. How the AI agent would look at the game is through the environment of the game which consists of 4 things:

1. The red block (the player controlled by the AI agent)
2. The blue circles which are moving around in a set pattern (which kill the player if they come in contact with the player)
3. The yellow coins which are spread out around the level and need to be compulsorily collected by the player (in higher levels)
4. The green area which is the goal which needs to be reached by the player after collecting all the coins in order to beat the level.

We initially thought about using Q learning to solve this problem, let's take a look at how Q learning would have worked in this case and why it fails to address our use case.

- **Q-learning**
  Q learning is an action-utility function, which can be used to evaluate the performance for taking an action in a specific state. Because in "the world's hardest game", the distance between the block and any obstacle, say L, is ranged in 0 and M_d, where M_d is the maximum length of the game map, by limiting a small number for each step, we only have finite combinations of the states and actions. Therefore, we can treat function Q as a table. In our Q table, each row records the reward for taking different actions in a state (Δe, Δw, Δn, Δs, Δd). As below:

State	E	W	N	S	Stay
(Δe_1, Δw_1, Δn_1, Δs_1, $\Delta d_1 s$)	1	10	20	-100	2
(Δe_2, Δw_2, Δn_2, Δs_2, Δd_2)	30	2	15	60	-1000
......	...	...	...	...	...
(Δe_m, Δw_m, Δn_m, Δs_m, Δd_m)	300	-40	55	40	-500

Table 1: Example Q-table in Our Problem

  This Q-table will contain m rows, which demonstrates the m states with utility values for each action.

  As a result, after finishing training, we will obtain a "perfect" Q-table. Here, "perfect" Q-table means every utility value will converge to a real number. To beat the game, we only need to find the corresponding row based on current state, and take the action with the largest utility value.

  This fails because our game has a Dynamically changing environment as the next "state" of our game is not solely dependent on the action taken by the player, it's also affected by the movement of the blue circles. Hence traditional Markov Decision Process (MDP) models fail to capture the entire dynamics of the game. Therefore traditional Q learning which is based on the MDP model fails to capture the entire dynamics of the game leading to fluctuation in values in the Q table using the update rule of the normal Q learning method.

- **Double Action Q (DAQ) learning**
  We use Double Action Q learning to help us with this issue. Double Action Q learning is different from a traditional MDP model (Q learning) in 2 ways:

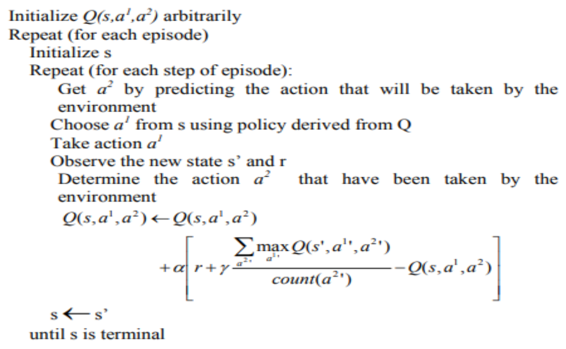
  1. The relationship and interaction between the red block - Player (agent) and the blue circles (obstacles) is represented by a new state. Each obstacle has its own set of properties and their own state with respect to the player. Final environment is represented as a summation of the state of each obstacle with respect to the player.
  2. The environment and the state of each obstacle can change without any interaction on behalf of the player. Both the player and obstacle can take an action independently which causes the change in states and hence a change in the environment. What the player observes is the final result after both the player and all the obstacles have acted.

  In DAQ learning, the Q table updates iteratively and the player "learns" to take appropriate action in different states. The updated rule for DAQ learning is

$$Q(s, a^1, a^2) \longleftarrow Q(s, a^1, a^2) + \alpha \left[r + \gamma \frac{\sum_{a^{2\prime}} max_{a^{1\prime}} Q(s', a^{1\prime}, a^{2\prime})}{count(a^{2\prime})} - Q(s, a^1, a^2) \right]$$

  Where a_1 is the action taken by the agent and a_2 is the action taken by the environment, count(a_2) is the number of a_2 available in the system, α is the learning rate, and γ is the discount rate.

- **PseudoCode**
  The algorithm was taken from Daniel C.K. Ngai and Nelson H.C. Yung (Ngai and Yung 2005)



```
Initialize $Q(s,a^1,a^2)$ arbitrarily
Repeat (for each episode)
    Initialize s
    Repeat (for each step of episode):
        Get $a^2$ by predicting the action that will be taken by the environment
        Choose $a^1$ from s using policy derived from Q
        Take action $a^1$
        Observe the new state s' and r
        Determine the action $a^2$ that have been taken by the environment
        $Q(s,a^1,a^2) \leftarrow Q(s,a^1,a^2) + \alpha \left[ r + \gamma \frac{\sum_{a^{2\prime}} \max_{a^{1\prime}} Q(s',a^{1\prime},a^{2\prime})}{count(a^{2\prime})} - Q(s,a^1,a^2) \right]$
    s $\leftarrow$ s'
    until s is terminal
```

Figure 1: The DAQ-learning algorithm

Results

All of our tests were carried out in a version of the game created on the Linux platform in Python3 using the Pyglet library. In the game, the state characterizing the Player-Obstacle interaction is represented by the x and y coordinates of the obstacle relative to the player (player position is considered as the origin). The player and the obstacles both can choose one of 5 actions:

1. Move up
2. Move Down
3. Move Left
4. Move Right
5. Stay at current position

The player is given a goal and obstacles are placed in the path according to the level specification of the Worlds Hardest Game. The player needs to traverse through the moving obstacles and find a path to the goal.

Each time a collision occurs, we restore obstacles and players to their original state and give a punishment to the agent. Since we have different levels of the game, we can use them to test the agent; and because we have the same type of the obstacles in different levels, we can use identical punishment every time a collision occurs. Furthermore, the density of obstacles are relatively fixed because the movement of them is bounded.

Also, in our experiment, We use the DAQ model instead of the Q learning model which leads to a lesser number of collisions and makes our model converges quicker. We also compare the number of epochs which both models need to converge, and show the result in a intuitive way.

- **Implementation**

  We implemented Double Action Q learning in our AI agent. In the learning process, we filled in a Q table, which provides a best moving choice based on current information, including the location of the red block and the location of all obstacles. Following are the main variables we need to construct, store, and detect in this process:

  1. s: indicates the current state of the red block, in our system, we construct a struct for s, containing 4 coordinates, each representing a corner of the block.
  2. a-a: indicates the action taken by the agent. This is a character variable in one of u, d, l, r, s, each of them denoting the movement of the block - move a step up, down, left, right or stay in place.
  3. a-e: indicates the action taken by the environment. In our game agent, the environment means the moving obstacles. So a-e is a vector of position of the moving enemy blocks.
  4. Q-table: a matrix which indicates the reward of each possible moving in each possible state
  5. r: reward of an action in a state.

  Our first step is using an open source library called “pyglet” to simulate the game. Since this part is a basic work to derive a possible environment to run our AI agent, but not significantly relevant about our algorithm, we will not discuss details in this part. The final behaviour of our game is the same as shown in “https://www.coolmathgames.com/0-worlds-hardest-game”, but simpler in following aspects:

  1. The gaming website allows the block to move in 8 directions(up, down, left, right, upleft, upright, downleft, downright), while our game only allows movement in four directions(up, down, left, right)
  2. There is a step limit for red block and obstacles, which indicates how far they can move in one action. The step limit can be manually changed in order to fit the training process.
  3. The x, y coordinates of the block and obstacles from the up-left corner can be easily stored for our agent system
  4. We only consider the map of the first level so far.

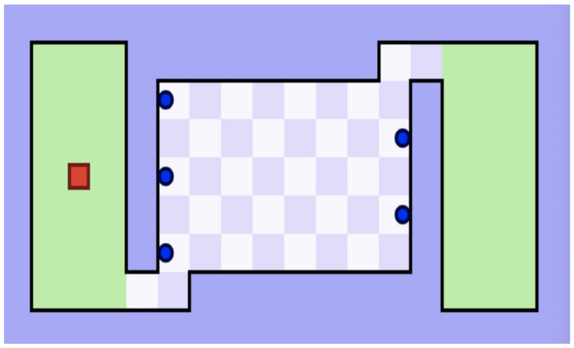


Figure 2: our simulated game

Then, we trained our system to obtain a converged q-table.Following are some important functions we implemented in our system.

1. load-initial: Load the game and initialize the location of the red block and obstacles
2. q-table-initial: randomly initialize the action we should take for all possible state and environment action.
3. get-environment-action: determine the next action of all obstacles based on a time t which is counted since we start the game, and their moving function.
4. get-agent-action: iterate among the Q-table to find the specific row of current state and environment action, choose the agent action with the highest reward and update
5. update-frame: take a-e and a-a as input, block move by a-a while obstacles independently move by a-e
6. get-reward: calculate reward by the current location of block and the given action
7. update-q-table: update q-table by the formula we used in methodology
8. DAQ-training: training our system in the following strategy using the pseudocode described earlier.

Finally, we pick different values for the number of epochs and study rate to see if we followed the action suggested by the q-table results from training, can we reach the goal and win the game. We will record the distance from destination in each situation and draw a graph.

- **Performance**

  We use different values of learning rate to train our system and record the cumulative reward for some epochs. We construct a line char as below which obviously demonstrates the result.
  In the line chart, we can conclude that learning rates with

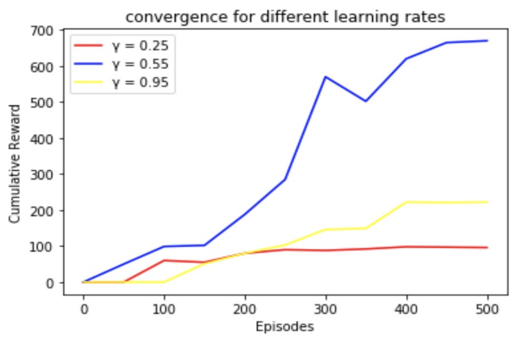


Figure 3: The performance of our model

too high and too low values both lead to cumulative reward vanishing problems. Learning rate around 0.55 is the most appropriate choice for our model. By setting learning rate equals to 0.55, our agent is able to beat the game in 600 rounds training.

- **Lessons**

First of all, we learnt how to use ”pyglet” library to simulate a simple game. Then, by researching many papers and models about reinforcement learning and Q-learning, we decided to use Double-Action Q-learning model to solve our research problems: build an AI engine to beat a popular platforming game called “The World’s hardest game”. In this process, we learnt how to use online resources to find relevant papers and quickly stretch their main idea.

To implement this, we firstly treated the block as a point at x, y coordinate and stored the position of the moving blocks as the information about the environment; however, this approach did not work because the agent did not track the walls (hence could get stuck trying to move against a wall) or had any info about the level layout. We reconsidered our DAQ model and were impressed by the online model of auto driving, and constructed a better approach: treat the player block as a real block with dimensions rather than as a point. We derived a class for the player block which contains four point coordinates to represent the four corners of the block. This class also has a function to obtain information about the environment by sending out lines in every direction from the body of the player block. By this approach, we implement a simple DAQ training algorithm which is able to beat the first level of “The World’s hardest game”. In the implementing process, we learnt how to divide a complicated model into several classes and functions. We also learnt when our model does not work, how to trace the problems and try to enhance our model.

Finally, we used different learning rates to train our system to see relationship between epochs and cumulative reword and figured the result. In this process, we learnt a basic relationship between the learning rate and the performance of our model.

Discussion

Looking at the results we can see that after 600 epochs with a learning rate of around 0.55, our AI agent using Double Action Q learning can successfully beat Level 1 of the Worlds Hardest game. Number of epochs required to train the agent is much lower with Double Action Q learning when compared to Traditional Q learning where it takes around 2000 epochs.

The agent trained with Double Action Q learning performs marginally better in new test levels than the normal Q learning agent.

This is in line with our hypothesis that in an environment with moving obstacles, Double Action Q learning agents perform better than traditional Q learning agents both in terms of training time and surviving and getting to the goal.

Our results agree with the previous research that we did especially with “Performance evaluation of Double Action Q learning in moving obstacle avoidance problem” - D.C. Ngai and N.H. Yung. The total number of epochs required to train and sum of negative reward was lower in Double Action Q learning when compared to Q learning.

Due to limited computational resources we had, we were unable to create an Agent which would beat the entire game. Our model generalizes very well when the obstacles are moving in linear motion with no left or right turns (backward turns are fine) but the performance starts suffering in terms of survival and number of collisions when we introduce objects moving in circular orbits. This can be attributed because we could not train the agent with levels containing obstacles with non linear orbits. Therefore our model can be used to beat most of the human generated levels with linearly moving obstacles.

Conclusion

In this paper, we focus on designing and training an AI model to beat an obstacles avoidance platforming game called “World’s hardest game”. To approach this, we did significant research around reinforcement learning, and inspired by the current AI models used in self-driving, we decided to apply Q learning. We implement both a traditional Q learning model and a Double-Action Q learning model. Depending on their behaviour, our Double Action Q learning model is able to beat the first level of “World’s hardest game” in less than 1000 epochs of training. We compared the training process and their behaviour under different learning rates, and presented them in a statistical way.

Our model has limitations in levels where obstacle movement is not linear or if they make any left or right sharp turns; for example, in the higher level of this game, obstacles move in circular motions rather than linear motion and our model shows significantly worse performance. So it would be interesting to try a different model which is able to beat games with non-linear moving obstacles and research on the reason for this improvement. Besides, we would also like to adjust the step-length of obstacles and allow players to move in eight directions rather than four to see how our model works in different situations.

References

Mihai Duguleana, G. M. Neural networks based reinforcement learning for mobile robots obstacle avoidance.

Ngai, D. C., and Yung, N. H. 2005. Performance evaluation of double action q-learning in moving obstacle avoidance problem.

Rahimi, P. G. P. . N. C. . S. Tuning computer gaming agents using q-learning.

Sallab, Ahmad EL; Abdou, M. P. E. Y. S. j. y. Deep reinforcement learning framework for autonomous driving.

Volodymyr Mnih, Koray Kavukcuoglu, D. S. A. G. I. A. D. W. M. R. 2013. Playing atari with deep reinforcement learning. *NIPS Deep Learning Workshop*.

Yung, D. N. N. Performance evaluation of double action q-learning in moving obstacle avoidance problem.