Collision Avoidance with Deep Reinforcement Learning

Raymond C. Li
Purdue University, li2026@purdue.edu

Jun Min Ahn
Purdue University, ahn60@purdue.edu

Zachary Tyler Esteron
Purdue University, zesteron@purdue.edu

Qiyin Hong
Purdue University, hong247@purdue.edu

Collision Avoidance with Deep Reinforcement Learning

Jun Min Ahn, Zachary Tyler Esteron, Qiyin Hong, Raymond Li

May 3, 2019

1 Abstract

In the past decade, learning algorithms developed to play video games better than humans have become more common. Google's DeepMind Technologies developed learning algorithms that could play Atari video games and also demonstrated their famous AlphaGo algorithm which outperformed professional Go players. However, little research has been done on learning algorithms developed to complete the particularly difficult single-player games. In particular, much further research could be done on developing learning algorithms for mechanically challenging games such as "bullet hell" games. We believe that agents could learn to efficiently evade obstacles utilizing deep reinforcement learning. The purpose of this study is to understand how to create such an efficient evasion algorithm. The deep learning model utilized is a convolutional neural network trained with a variant of the Q-learning algorithm. The model is given positional coordinate data and bullet location data as its input and outputs a value function to determine the best following action. The agent controls the directional inputs of the game's user avatar and its inputs, or actions, are modeled as a two-dimensional Markov decision process. The agent uses the game's internal score and the amount of time its user avatar avoids being hit by obstacles as its target and experiments with its inputs each episode to increase the maximum reward. Each training episode is reset when the user avatar is hit by the bullets.

2 Introduction

Reinforcement learning algorithms are developed to understand the best answer to an environment. The environment is the world which the agent, the algorithm, "sees". Within this context, an ideal environment would be one where the agent is given all information about the game that a human player would have access to. Just as humans can see all the bullet locations and their trajectories, an agent would ideally have access to the same information. Then, the agent would utilize a reward

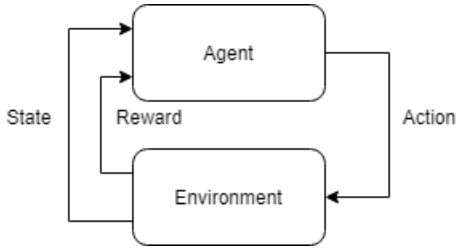


Figure 1: Block diagram of Reinforcement Learning algorithm.

system developed by the programmer to decide on its best next action. The reward system takes feedback from the agent's actions and is returned by the environment [1] [3]. The reward is the value that the agent tries to maximize which is the score in the game.

3 Objective

Touhou is a two-dimensional bullet-hell game in which the main objective is to traverse each level while avoiding various dangers such as enemies and the bullets they shoot. The problems tackled by this project, therefore, involve developing an agent that can act in different situations and transforming the game state into an environment that an agent can read and process. The objective of this project is to develop a learning algorithm that is capable of efficiently evading within the game and thereby maximize the score achieved by the agent and increase the time the agent can survive in the game.

4 Methods

4.1 Setup - Developing the Environment

When attempting to develop the environment for our reinforcement learning algorithm, we first used an application called Touhou Toolkit to extract the game files and change the sprites to have uniform colors. We changed enemies and bullets into black, we changed the background to white/gray, and the agent hitbox to red.

Using a program called Cheat Engine, we were able to locate the value for position data for the agent, death counter, and the current score of the agent in memory. After a few trials, we found that the position of the agent and the deaths were always a fixed offset from the program call and the score would always change. We decided not to use the score as the reward, not only because we would have to relocate the score using Cheat Engine after each run, but also because the score would change based on many factors in the game such as where the agent was, the time it took for the agent to destroy the target, whether the agent was able to pick up a power-up that was

(a) Before (b) After

Figure 2: The sprites of the game before and after the sprites were changed for easier analysis by a computer program.

Address	Value	Previous
004BDCA0	276	287.9656372
004BDCA4	320	362.7107544
004BDEEC	252	263.9656372
004BDEF0	320	362.7107544
004BE138	300	311.9656372
004BE13C	320	362.7107544
004BE408	244	255.9656372
004BE40C	304	346.7107544
004BE420	243.1750031	255.1406403
004BE424	303.1749878	345.8857422
004BE42C	244.8249969	256.7906494
004BE430	304.8250122	347.5357666
004BE438	242.6000061	254.5656433
004BE43C	302.6000061	345.3107605
004BE444	245.3999939	257.3656311
004BE448	305.3999939	348.1107483
004BE450	232	243.9656372
004BE454	292	334.7107544

Figure 3: Memory locations of the player's position

dropped, etc. This would cause the agent's score to vary by a great deal even if the agent took effectively the same path. Since our main goal in this project is to teach the agent to evade, not to maximize the score in this game, it would be detrimental to the agent's learning to use the game's score as its primary reward function.

After figuring out the memory locations, we used a python library called ReadWriteMemory to read the memory at those specific addresses.

The next step was to find ways to control the game. To do that, we used the ctypes library to simulate direct key inputs from the keyboard. Using these direct inputs, we were able to code macro keys to start and restart the game and allow our python program to fully control where the agent goes. We could also change the speed at which the game would run so that our algorithm had sufficient time to analyze and submit key inputs. In each of these methods below, we gave the agent two actions to choose from, to move either left or right.

```
ProcID = rwm.GetProcessIdByName('Th07.exe');
hProcess = rwm.OpenProcess(ProcID)
x = rwm.ReadProcessMemory(hProcess, 0x004be498)
deaths = rwm.ReadProcessMemory(hProcess, 0x01345e3c)
```

Figure 4: Functions to grab memory information

```
def pressKey(hexKeyCode):
    extra = ctypes.c_ulong(0)
    ii_ = Input_I()
    ii_.ki = KeyBdInput( 0, hexKeyCode, 0x0008, 0, ctypes.pointer(extra) )
    x = Input( ctypes.c_ulong(1), ii_ )
    ctypes.windll.user32.SendInput(1, ctypes.pointer(x), ctypes.sizeof(x))

def releaseKey(hexKeyCode):
    extra = ctypes.c_ulong(0)
    ii_ = Input_I()
    ii_.ki = KeyBdInput( 0, hexKeyCode, 0x0008 | 0x0002, 0,
ctypes.pointer(extra) )
    x = Input( ctypes.c_ulong(1), ii_ )
    ctypes.windll.user32.SendInput(1, ctypes.pointer(x), ctypes.sizeof(x))
```

Figure 5: Functions to to simulate direct key inputs

4.2 Method 1 - Proximity Detection

We had our environment capture a portion of the screen to analyze the game visually. After capturing a portion of the screen, a red mask was run through the image to filter for the agent and a black mask was run to filter for the bullets. Through a Hough Circle Transform, we were able to find the agent's center and transfer that data to the agent. Then, a proximity detection algorithm was used to detect bullets near the agent.

The proximity detection algorithm draws lines coming out of the agent. The agent is given the maximum reward if no bullets interrupt the lines coming out of the agent. If a bullet interrupts the line, then the reward is decreased by the distance the lines is interrupted and the agent would act accordingly. A total of 16 lines are drawn around the agent. By creating such a reward system, we believed we could create a system that would incentivize the agent to move away from the bullets so as to maximize the reward gained.

To train this agent, we started the exploration rate at 1 and decayed this rate by a factor of 0.995 during every single generation with the limit set at 1000 generations. This way, the agent would start out by making more random actions and after testing certain strategies, it would converge to the best method that was found during learning. If we found that it wasn't converging to a solution that is optimal, then we would restart the program, again with a higher exploration rate.

Figure 6: The proximity detection algorithm draws lines coming out of the agent and shows the closest bullet (indicated by a green line) to the user.

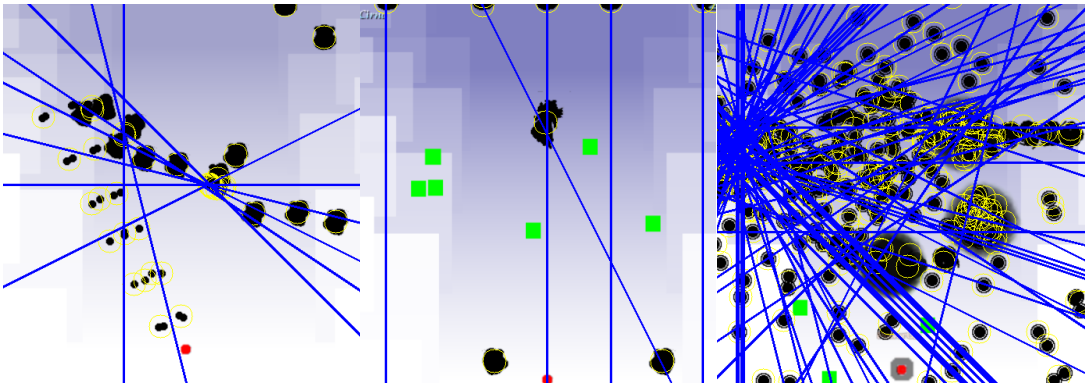
Results: For this specific trial, we ran the game at full speed. The computer we used could grab an image and process it about every 17ms on this algorithm. The test length was set at 50. After about 14 hours of training and 600 episodes, the agent had learned to dodge and could consistently score about 20% better than when it first started but had trouble getting further. The agent had learned to stay still by alternating left and right. The agent had also learned to stay next to the walls so that the dodging becomes easier, but still had trouble dodging the faster bullets.

We suspect that the reason that it had trouble dodging the faster bullets is because there wasn't enough time for the agent to predict the bullet trajectory and react accordingly. In order to fix this, we could either slow down the game or run this reinforcement algorithm on a faster GPU and the images would be analyzed at a faster rate. We could also extend the test length so that the agent can see the bullets coming in sooner. Another issue was the perception of time on the agent, although we could obtain images at a fairly consistent rate, a slight difference (3ms) in timing may affect what the agent decides to do. Using a faster GPU, we would be able to set a limit for the frame rate to match the game.

4.3 Method 2 - Bullet Prediction

We started off by capturing a section of the screen to analyze the trajectory of the bullets and enemies on screen. Then we used OpenCV's contour functions to determine the location and the quantity of the enemies on the screen. We would compare frame by frame to determine the trajectory of the bullets so that the agent could dodge more efficiently. The trajectories of the black elements on the screen (theoretically the bullets) would be stored as a slope and converted into the state that the agent analyzes. If the agent was in the way of one or more of the trajectories, the agent would receive a negative reward based on how far the object is, otherwise, it would receive a positive reward.

Results: For this specific trial, we ran the game at full speed. The computer we used could grab an image and process it about every 10-400ms. The speed at which the image is processed depends on how many lines are generated from trajectory



(a) (b) (c)

Figure 7: Blue lines are calculated trajectories

slopes. This tracking algorithm made use of the centers generated by the contour algorithm in OpenCV as well as a localized proximity algorithm. Because the centers generated are slightly inaccurate (sometimes multiple centers would be found where one center should be or a previously found center would be analyzed to be stationary), the prediction lines that are made are not reliable at all. Figure 7 shows what these trajectories look like. In the Figures 7a and 7c, most of the detected contours don't have a slope associated with it because it has not moved since the last frame. Figure 7b shows when all of the black elements are moving at a speed high enough for the algorithm to see the change in centers.

Our attempt to fix this was to use Kalman filters along with the centers but the code was not optimized for tracking a changing amount of objects on the screen. This filter would run would confuse different objects as they disappear and reappear on screen.

4.4 Method 3 - Image Stacking

As before, we had our environment capture a portion of the screen to analyze the game visually. However, for this method we used the same method created by Thomas Simonini that was used to develop an algorithm to play the game Doom [2]. However, for this method we had the environment capture a portion of the screen four times and convert those four frames into grayscale images. Then we created a single stack of those four images and passed that stack as an input into the Deep Q Neural Network. The frames were processed by three convolution layers each using an Exponential Linear Unit activation function. A fully connected layer with ELU activation function and another fully connected layer with a linear activation function produced the Q-value estimation for each action. On top of this, we stored observed experiences in a reply memory and select a small batch of these memorized experiences and learned from that batch using a gradient descent update step.

Results: When running it on our own computer, we found that this algorithm takes too long to run. One action takes about 5 seconds to run. If we were to have the agent make a decision every single time the game changed, we would have to slow the game to a crawl and run the game for about 2 hours per episode. Upon further inspection, we found that only the learning part of the algorithm (shown in Figure 8) takes this amount of time because it has to sort through all of the frames. A solution would be to switch to another dedicated GPU so we can run it faster.

```
# Get Q values for next_state
Qs_next_state = sess.run(DQNetwork.output, feed_dict = {DQNetwork.inputs_: next_states_mb})

# Set Q_target = r if the episode ends at s+1, otherwise set Q_target = r + gamma*maxQ(s', a')
for i in range(0, len(batch)):
    terminal = dones_mb[i]

    # If we are in a terminal state, only equals reward
    if terminal:
        target_Qs_batch.append(rewards_mb[i])

    else:
        target = rewards_mb[i] + gamma * np.max(Qs_next_state[i])
        target_Qs_batch.append(target)


targets_mb = np.array([each for each in target_Qs_batch])

loss, _ = sess.run([DQNetwork.loss, DQNetwork.optimizer],
                   feed_dict={DQNetwork.inputs_: states_mb,
                              DQNetwork.target_Q: targets_mb,
                              DQNetwork.actions_: actions_mb})
```

Figure 8: The learning section of the Image Stacking algorithm.

5 Conclusion

Moving forward, further research could be conducted to improve the processing speed for the development of the environment and optimization of the reward function to improve the algorithm. A way to save the progress or the reward function so as to not have to repeat the program from episode 0 should also be added to allow for transfer learning opportunities. Finally, research could be done to determine whether the processing delay resulting from an algorithm that predicts bullet patterns is negligible in comparison to the value such an algorithm could bring to the agent's evasion efficiency and accuracy.

6 Next Steps

With this research as a foundation, we hope to create a more efficient prediction algorithm to predict the trajectory of the bullets in the game. Perhaps creating a separate neural network for Touhou to track bullets might be a viable solution. We

also hope to test pre-existing object tracking and prediction algorithms to test their efficiency in improving the algorithm in this game. To improve image detection and tracking, it may also be useful to refine the sprites. The reward function may need to be further optimized to allow the agent to learn more efficiently. We could also further optimize the exploration rate by creating a feedback loop so that it could learn more by itself. Finally, we hope to be able to move the game onto a Linux based system so we can use services such as Google Collab and Scholar for research computing; this will allow us to run the game at a higher speed and test the last method.

References

[1] David Silver. *Reinforcement Learning*. URL: `www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching.html`.

[2] Thomas Simonini. *An Introduction to Deep Q-Learning: Let's Play Doom*. URL: `medium.freecodecamp.org/an-introduction-to-deep-q-learning-lets-play-doom-54d02d8017d8`.

[3] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. 2018.

7 Acknowledgements

Thank you professor Guang Lin, professor Carla Zoltowski, and Lusine Kamikyan for making this project possible.