AI for Doodle Jump

Chunhao Bi
2019533135
bichh@shanghaitech.edu.cn

Sizhe Gu
2019533200
guszh@shanghaitech.edu.cn

Yichen Zhou
2019xxxxxx
@shanghaitech.edu.cn

Abstract

Doodle Jump is a casual but addictive mobile minigame in which the player moves the doodler left and right, jumping up between different types of platforms. There is a challenge of reaction and operation for players that they need to keep doodlers from falling into the void in the increasingly difficult game. This paper then considers whether AI can be trained to perform this challenge. Therefore, we focused on porting the mobile game to the PC as a Python file, and we trained our doodlers in two different ways. At the end of the paper, we analyze the advantages and disadvantages of these two methods and compare them.

1. Introduction

Doodle Jump is a casual but addictive mobile minigame that launched in March 2009 and hit the 10 million download mark in 2011. Back when mobile games were just getting started, it set off a trend. And it remains fun and influential to this day.

2. Reinforcement Learning

3. Q-Learning AI

Q-Learning is a traditional method for Reinforcement Learning. We have already implemented this method in the Pacman game introduced in our lecture. Despite that the result is not good when facing several ghosts, the pacman can achieve a quite satisfying behaviour. In the doodle jump game, we also tried to implement Q-Learning for decision-making, and achieved a quite good result. In this project, the approximate Q-Learning method is used, taking reference of lecture project 5.

3.1. State Representation

The state representation in pacman searching problem is quite easy, when the state space is small enough to be stored. However, in Doodle Jump, the state is very complicated as there are many platforms. Therefore, only the approximate Q-Learning using features can be used.

The state mainly contains the player, the platforms that can be step on, and even the monsters and springs. The actions are moving left, moving right and staying still. In each call of getQvalue(), the player's position is predicted similarly used in moving part, as an expression of action. The feature extractor used the predicted position to calculate the Q value of the current environment. When the game updates, the real Q value is used to calculate the time difference and update the weights. This is done in update().

3.2. Feature Extraction

In this part, some useful features are extracted for evaluating Q value of each state and action.

3.2.1 Platforms

The platforms have different importance considering the different distance and height from the player. A general solution is to consider a platform that is above the player, which is most likely to be reached. This makes the player jumps slowly but safely. However, considering one platform is not enough, for that in some cases, the nearest above platform is away from the others. It is more likely for a human player to choose the direction that have more platforms, which is safer if he missed one step. Therefore, several platforms above are considered as features.

To get a good result, the platform choice is complicated. Below is a some reference code.

```
# below is some codes of choosing part
# self is the player
  for p in platforms:
      ...
      # the below one is not too far
      if (abs(p.x − self.x) < 300):
         tmpCandidates.append(p.x)
```

```
...
# check the p above is not too high
if ( self .startY + maxHeight < p.startY):
    break
...
# check the p above is reachable
isRight = (p.x − self .x) > 0
farestDist = \
self .jump∗(self.xvel∗2+self.jump∗isRight)
if (abs(p.x − self .x) < abs(farestDist )):
    returnPlatforms.append(p.x + p.vel)
```

To be noticed, only the platforms that are not too high, and are reachable after a whole jump, can be added into the feature. The tmpCandidates stores the platforms below the player. When the player is falling, we don't want it to head for those above. On the contrary, it focuses on those below for landing. These features are represented as possible platforms, which are the platforms the player intends to head to.

3.2.2 Monsters

The monsters can also be used as features. However, because the monsters are not regularly placed than platforms, its weight is not very useful. In our test, there is not much difference when monster feature is added. A probable reason is that because the weights are not updated simultaneously with platform weights when the monsters don't appear. So the old monster weight may not be influential when new platform weight are updated.

3.2.3 Distance

The distance from player and platforms is directly related to the feature value. Intuitively, player will choose a near platform. He prefers a step right when a right platform is closer than a left one. Thus, when the distance change are the same, a nearer target platform needs a bigger Q-value difference to make the agent choose the action. Therefore, we took the log distance, in which $Q - value$ varies smaller when $distance$ are bigger. The agent will choose the nearer one when two platforms have the same weights.

```
feats [ 'firstUpDist' ] = math.log2(
    abs((upPlatforms[0] − player.x)/600)+1)
```

3.3. Results

A demo training record is here. Not like Generic Algorithm which uses neural network and update once each game. The player agent updates its weight every game update. So in the first game, the player can already achieve a quite good score. To be observed, the score have a slight rise in the second and third game. On average, the score can reach 25000+ after 3 or 4 games.

Actually, due to the simpleness of Q-Learning where the features have linear relation, the agent can hardly achieve as good result as using neural network. We can observe that in several complex cases the agent may make a stupid decision. However, as an AI it already achieves a good behavior.