Swing Copters AI

Monisha White and Nolan Walsh

mewhite@stanford.edu | *njwalsh@stanford.edu*

Fall 2015, CS229, Stanford University

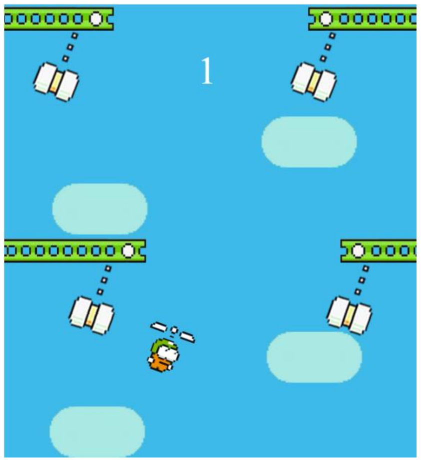
1. Introduction

For our project we created an autonomous player for the game Swing Copters.

1.1 Swing Copters

The goal of Swing Copters is to keep the avatar alive as long as possible. The avatar dies when it collides with a platform, a hammer, or either edge of the screen (which we will call a "wall"). A player has one way to control the avatar's trajectory – tapping the screen to reverse the avatar's acceleration.

Despite being a simple game, it is surprisingly difficult to play. Beginning players can rarely even make it past the first platform.



Left:

A screenshot of Swing Copters gameplay

1.2 Motivation

Video games provide a great testing ground for machine learning algorithms and techniques. Consequences of failure are low, and many games pose unique and unanticipated difficulties that would not occur in the real world.

Swing Copters is a good candidate for a machine learning project since it is very difficult for humans to play well. Without many hours of practice, people have difficulty navigating through even a few consecutive platforms.

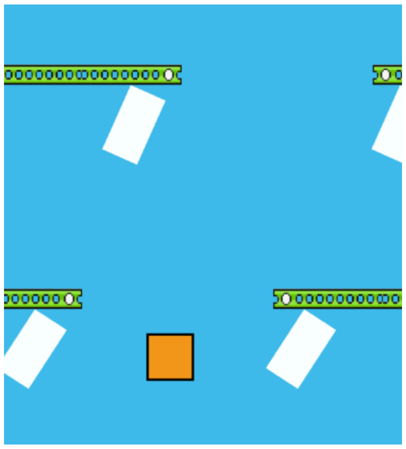
1.3 Setup

The input to our algorithm is the current game state – the positions of the avatar, the swinging hammers, and the platforms, the velocity of the avatar and angular velocity of the hammers, and the acceleration of the avatar. We used MCTS and Q-Learning to create players who would then output the player's chosen action – 'click' or 'don't click' (corresponding to negating the avatar's acceleration or not).

We implemented our own version of the game to use for training and testing our algorithms. We also implemented a modified version of K-Means Clustering to extract information about the game state from screenshots of the original version of the game. This serves as a proof-of-concept that our player could be used to play the game on a mobile device or emulator. The input for this problem was a screenshot of the game, and the output was a location (x, y coordinates) of the avatar's location in that screenshot.

Right:

Our version of Swing Copters



2. Related Work

Computer games are a popular area for machine learning algorithms. In 2013, engineers at Google released a paper detailing a system called "DeepMind." This system used a combination of Q-Learning and a neural network to play old Atari games with great success (Mnih, Kavukcuoglu, Silver, Graves, & Antonoglou, 2013). This paper motivated much of our work on Swing Copters even though we did not end up using a version of their Deep Q Network algorithm.

There are several instances of people detailing methods creating autonomous players for Flappy Bird, a similar mobile game (Gatt 2013; Vaish 2013). All implementations we found use some form of Q-Learning. This is a reasonable approach since Q-Learning lends itself to work in video games in general. In addition to playing games created for humans, Q-Learning is also being used as a way to control the behavior of non-player characters in new games (Patel 2009).

We only found one other implementation of a learning algorithm for the game Swing Copters. That project also used Q-Learning, but performed their research on a simpler version of the game without the swinging hammers (Miller, Beiser, 2014). Our work improves upon their results by using a full model of the game.

3. Dataset and Features

3.1. Q-Learning Player

As input, our algorithm takes a single complete game state. All values needed to describe the game state are discrete (on the order of pixels). In order to completely describe a single state of Swing Copters, we give our algorithm the positions of all game objects (hammers, platforms, and the avatar) along with their respective velocities and the avatar's acceleration. We then compute features on the game state.

The biggest challenge to our approach was coming up with useful features. We considered many different ways to combine aspects of the game state:

– Product of avatar's velocity and acceleration
– Product of avatar's distance to gap between platforms and acce1eration (with differing velocities)
– Avatar's x distance to the platform it's facing
– Avatar's x distance to the nearest wall
– Avatar's x distance to the wall it's facing
– Product of avatar's distance from center, velocity, and acceleration
– Avatar's minimum stopping distance from wall
– Avatar's minimum stopping distance from platform
– Indicator on whether or not the avatar had enough time to make it to the gap between platforms
– Product of Avatar's signed x distance to the gap between platforms and acceleration

Each of the continuous features above was discretized into indicator features on ranges of their values before being given to the Q-Learning algorithm.

The biggest challenge for feature selection in our implementation, was that to be successful, a set of features needed to be able to differentiate between two very similar states. At any point, a Swing Copters player can choose between two states: one where the avatar is accelerating to the right, and one where the avatar is accelerating to the left. The first features that improved our results significantly were indicators on the avatar's "stopping distance" from a wall – how close to a wall the avatar would be if the player started accelerating in the opposite direction immediately. We computed this stopping distance from the wall that the avatar was moving towards. These features allowed our player to learn how to avoid crashing into the walls (sides of the playing area). Computing similar features on the avatar's relation to platforms and hammers were also helpful.

3.2 K-Means Avatar Detection

For the avatar detection portion of our project we took 169 screenshots of gameplay on the original version of Swing Copters. We then compressed each screenshot down to a 289 by 314 pixel image in order to lower the runtime of the algorithm. Next, we annotated the avatar's location (x, y coordinates) for each screenshot. This formed our test set.

4. Methods

4.1 Monte Carlo Tree Search

Before we built our Q-Learning player, we implemented a player that uses Monte Carlo Tree Search (MCTS).

One difficult aspect of Swing Copters is that the player can only control the direction of the avatar's acceleration. All actions produce magnified differences in the avatar's future position. We wanted to factor in these delayed changes in position by seeing how the game would play out given a particular action. MCTS is well suited for this because it estimates the reward of a particular action by performing a series of simulated games.

Each iteration of MCTS has four phases: selection, expansion, simulation, and backpropagation.

1. In the selection phase, we select an unexplored state of the game's state tree: *s*.
2. During expansion we expand the state tree to include all states that could result from actions taken from *s*.
3. During simulation, we play a random game starting from *s*.
4. Finally, in backpropagation we update the predicted values of each state along the path from *s* to the root of the state tree.

Another aspect of Swing Copters that makes MCTS a worthwhile approach is that consistent short term survival leads to success. A player using MCTS computes the expected reward by averaging the randomly played games. In Swing Copters, although random play will not lead to long term success, it can give a good idea of the likelihood of short term survival.

4.2 Q-Learning

Q-Learning is a reinforcement learning technique that updates a model using state, action, reward, next-state tuples in order to try and learn the best action value function. For our project we implemented Q-Learning using a linear function approximation model as follows:

- Define features $\varphi(s, a)$ and weights $\mathbf{w}$
- Define $Q_{opt}(s, a; \mathbf{w}) = \mathbf{w} \cdot \varphi(s, a)$

On each (s, a, r, s'):

$$\mathbf{w} = \mathbf{w} - \eta[Q_{opt}(s, a; \mathbf{w}) - (r + \gamma * V_{opt}(s'))]\varphi(s, a)$$

Here Q_{opt} is the current best estimate of the value of a state, action pair. Linear function approximation means that we estimate Q_{opt} using the inner product between the current weights **w** and the feature vector $\varphi(s, a)$, as in linear regression. From each (s, a, r, s') we attempt to update **w** so that future predictions better match the true value of an action. Using our learned Q_{opt} from each state we choose the action that has the highest estimated value.

4.3 K-Means Avatar Detection

K-Means is an unsupervised learning algorithm that finds clusters in data.

1. Randomly initialize a set of cluster centroids $\mu_1, \mu_2 \dots \mu_k$
2. Repeat until convergence:

   For each *i* set: $c^{(i)} := \arg\min_j ||x^{(i)} - \mu_j||^2.$

   For each *j* set: $\mu_j := \frac{\sum_{i=1}^m 1\{c^{(i)} = j\}x^{(i)}}{\sum_{i=1}^m 1\{c^{(i)} = j\}}.$

K-Means alternates between assigning every data point to the nearest centroid, and updating each centroid to the average of all points assigned to it. The algorithm converges when all points remain in the same clusters

Our goal with K-Means was to take advantage of the fact that different objects in Swing Copters are colored differently.

K-Means Avatar Detection:
Input: Screenshot of Swing Copters

1. Assign μ_1 = average color of avatar
2. Assign $\mu_2 \dots \mu_k$ = color of random pixel in the image
3. Run K-Means to convergence keeping μ_1 constant

Output: The average location of pixels assigned to μ_1.

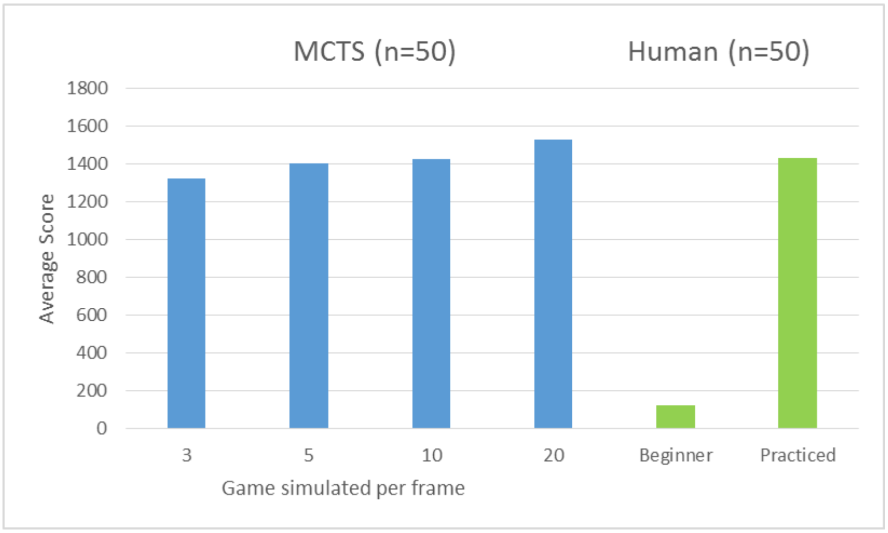
The same algorithm can be extended to find the location of the hammer and platform objects. First we initialize μ_1 to be the average color of the object we want to detect. Then, instead of returning the average location of pixels in that cluster, we perform K-Means on the location values of those pixels. This gives a set of clusters, with each one centered around one instance of the object being detected. At any point in Swing Copters there are either four or six platforms and hammers. Thus, we can run this algorithm with K = 4 and K = 6 and pick the result with the tightest clusters.

5. Results

In order to measure the success of a player, we performed a series of trials and averaged the number of frames of the game that the player survived.

5.1 MCTS

The MCTS algorithm produced a mediocre Swing Copters player. The player regularly navigated through a few platforms, already better than a beginning human player. However, it was not consistently successful, and rarely scored better than a practiced human player. Scores improved slightly with more simulated games. However, the computation time required increased drastically, making it impractical as a way to play the game in real time.



5.2 Q-Learning Player

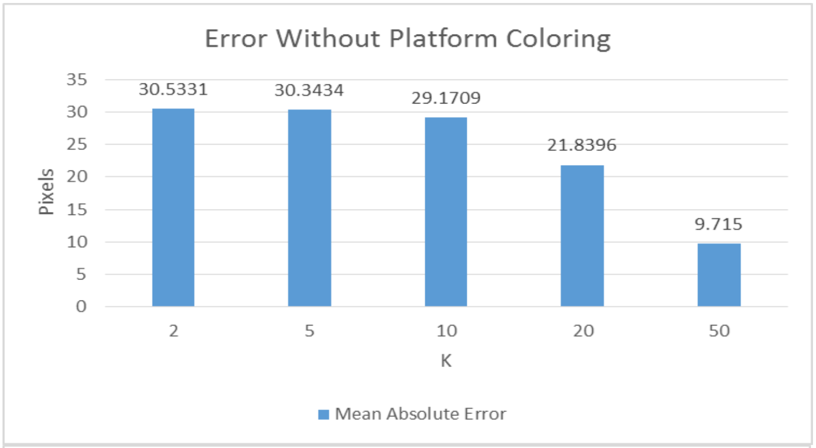
The quality of the Q-Learning player depended heavily on the set of features it used to learn. As we discovered more useful features, the player was able to avoid more objects and survive longer. Using our full set of useful features, our player can survive indefinitely.

Features	**Survival Time (frames) (n=50)**
stopping distance from wall	577.45
stopping distance from wall + platform	1297.30
stopping distance from wall + platform (multiple distances) + vertical space	indefinite
Human Player	
Beginner	124.34
Practiced	1430.32

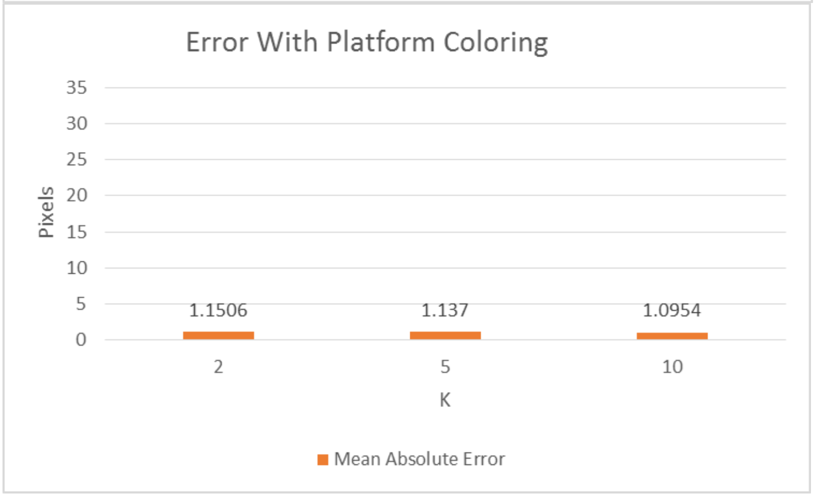
For our Q-Learning player we tested different exploration probabilities, and discovered that a value of zero worked best. There is enough randomness inherent in the game that many different states were still explored. At the same time, the inputs required to be successful in Swing Copters are precise, so choosing a random action will often lead to dying shortly after.

5.3 K-Means Avatar Detection

We performed our K-Means Detection algorithm on the set of annotated screenshots using different values for K. We calculated our error as the average number of pixels our prediction differed from the annotation.



As K increased we saw a decrease in our error. We realized that this was because with higher values of K, it was more likely that one of the centroids was initialized to be the color of the platforms (the closest color to the color of the avatar).



By initializing one of the centroids to the average color of a Swing Copters platform, we were able to bring the error down to less than two pixels.

6. Conclusion

We are very impressed by the results from our Q-Learning player. Using the top performing set of features, the player learns to survive indefinitely. The Q-Learning player far out-performed the MCTS player and practiced human players. While we saw that MCTS continued to improve its play as we increased the number of simulated games, doing this also decreased its speed to below acceptable performance. The Q-Learning player only needs to perform two vector multiplications between the weights and the features of the two possible state action pairs in order to decide which action to take. This leads to lightning fast performance. We were also pleased by the success of the K-Means algorithm for detecting the avatar’s location. An error of less than two pixels can create an accurate enough game state to successfully play the mobile version of the game. With more time, we would like to extend the algorithm to detect the locations of hammers and platforms. Then we would be able to complete our original goal of creating a player that plays the original version of the game on a mobile device.

References

Gatt, Brian. "Reinforcement Learning - Flappy Bird." Goldsmiths University of London, 2013-2014. <http://www.brian-gatt.com/portfolio/flappy/documentation.pdf>.

Miller, Jared, and Chris Beiser. "Creating Autonomous Swing Copters Using RL Techniques." 12 Dec. 2014. Web. 10 Dec. 2015. <https://github.com/cadtel/QCopters/blob/master/QCoptersReport.pdf>.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., & Antonoglou, I. (2013). Playing Atari With Deep Reinforcement Learning. NIPS

Patel, Purvag. "Improving Computer Game Bots' Behavior Using Q-Learning." Southern Illinois University Carbondale, 1 Dec. 2009.

Vaish, Sarvagya. "Flappy Bird RL." 2013. <http://sarvagyavaish.github.io/FlappyBirdRL/>.