Reinforcement Learning Exploratory Analysis: A Million Kirby's Fail at Walking

Stephen Yu : 405570842[a]

[a] *University of California, Los Angeles,*

1. Introduction

The overall goal of this project was use reinforcement learning to implement a machine learning agent that can progress through a Kirby-like 2D platformer game. Using reinforcement learning methods such as a present state stimuli, a reward function, and a genetic algorithm the agent can, over time, learn from its previous mistakes and eventually complete the level. However, to accomplish this we first needed to create the game the agent would learn to beat using Python package pygame, which in particular must include a well-defined win condition. This brings us to the design of the game itself.

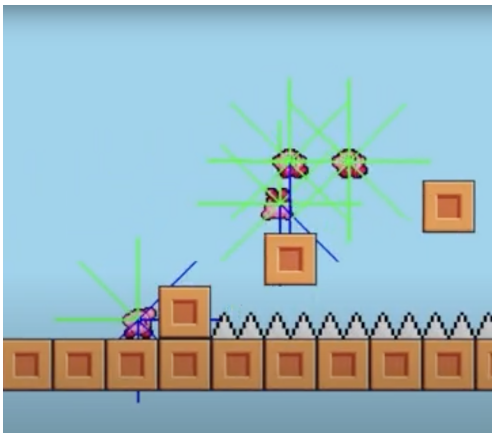

Figure 1: Kirby Learning to Beat Level Four

2. The Game

A Million Kirby's Fail at Walking, as it is affectionately called, is a rudimentary platformer with a standard control scheme. The player can move left or right, and can jump once after having touched the ground. In addition to the player object and terrain blocks with collision, there are two hazards: spikes, and the void (each of which mark the player as dead upon collision). Each level also includes one flagpole at the end, which marks the player as having won if they collide with the flagpole object at any point.

3. Methods

A successful reinforcement learning algorithm is extremely similar to some evolutionary concepts such as survival of the fittest and mutation. Reinforcement learning essentially boils down to learning by failing a lot and succeeding a little thus the name of the project: "A Million Kirby's Fail at Walking". Before going into each method in detail let's dictate the overall process a reinforcement learning agent uses to train and beat levels.

3.1. 10,000 Foot View

The agent, Kirby, will make decisions based off of present state stimuli being fed into as inputs into a neural network, and the output with the greatest weight will be the decision Kirby will make (go left, go right, or jump). Present state stimuli encompasses information about the Kirby at a particular frame such as it's x coordinate or distance to nearest obstacle, etc. At the beginning, also known as the first generation, the model's parameters (weights and biases) are set completely randomly.

There will be 10 different randomly generated models, and the model that reaped the greatest reward from the reward function will be kept. A reward function outputs a scalar that takes into account how "well" a model has performed; a Kirby that gets closer to the flag and avoids death will have a higher reward. From this, the model with the highest reward out of the 10 will be propagated and mutated to create 10 new children which form generation two. Of these children, the two models with the greatest reward will be kept, and the cycle continues until Kirby has beaten the level.

3.2. Present State Stimuli (Inputs)

- Character Position (2)
- Flag Position (2)
- Character Velocity (2)
- Jump Availability (1)
- Sightline distance (8)
- Sightline object type (8)

Note that the value in the parenthesis is the number of neurons in the input layer attributed to that piece of information. In this case, the position/velocity vectors are split into their x and y components, and there are a total of eight sightlines evenly distributed around the player. For a further explanation of the sightline data, distance records the distance of the nearest object along the line's direction, and object type records the class type of that object (block, spike, flag).

3.3. Reward Function

The reward function is dictated below:

- W = Whether the model won or lost (1 for win, 0 for loss)
- D = Net x distance traveled by the model
- T = Time until completing the level
- Death = Whether or not the model died (1 for death, 0 for survival)

The score S for the model is calculated as:

$$S = 1000 \times W + D - T - 50 \times \text{Death}$$

This result of the 10 models in any given generation would be ran through this reward function to calculate how well each of the models did. Then, the model with the highest reward would be kept for the next generation while the other eight models are scrapped.

As seen above, the reward function gives positive weight to more effective and efficient actions from Kirby and negative weights to the contrary. You notice an overwhelming positive weight of 1000 for a model that reached the flag while death only carries a negative weight of fifty. The reason for this is that the learning sometimes ran into local maximums where the agent would rather stay far away from the flag and not die, than explore and make progress towards reaching the flag. The two middle variables, distance and time, are logical as a model that gets closer to the flag should be kept over a farther model, and a model that takes shorter to complete the level should be kept with all other variables being held constant.

3.4. Propagation and Mutation

The mutation and propagation functions are both crucial components of the genetic algorithm, which allow for the formation of a new generation and improvement of the model over time. Our propagation function uses the fact that a neural network can be fully defined by its weights, which can be interpreted as the "genes" of the model. When a child model is ready to be formed, each weight/bias layer of the child is randomly chosen to adopt all of its values from one of its parents. this process then repeats for every layer of weights and biases, completing one offspring. After the chosen number of children are generated, each child passes through the mutation function. There are many possible ways to implement such a function, but we found that the multiplicative approach works best, which is as follows. Once a model is passed to the mutation function, each of its individual weights rolls a 50 percent chance of remaining unchanged. If a weight is chosen to me mutated, its value is then multiplied by a random variable chosen from a normal distribution with mean one and standard deviation of one half. We found that this multiplicative approach trains faster, as it takes a smaller number of favorable mutations to reach very large

or very small weights. In addition, there is a small probability that a weight changes its sign, though this only occurs past two standard deviations down. This combined with the Relu activation function allows the model to send certain inputs to zero more easily, which helps with differentiating useful versus non-useful input data.

4. Models

There were a few variables in the model's structure that we experimented with in an attempt to optimize the agent's performance. Components of the model's genetic algorithm, such as the reward function and mutate function, as well as the number of hidden layers in our neural network are easily customizeable. Changing our reward function to place less value on death and more value on distance resulted in a substantial improvement of Kirby's performance on later levels with spike obstacles.

The most effective manipulation, however, is easily the input variables of our model. We experimented with three main manipulations of input variables. The most elementary model takes in only basic self and flag location, ability to jump, as well as character velocity information. Next, the most complex model takes in the same basic information as the previous model, with the added input of sight lines. Finally, the last model considered only sight information.

5. Results

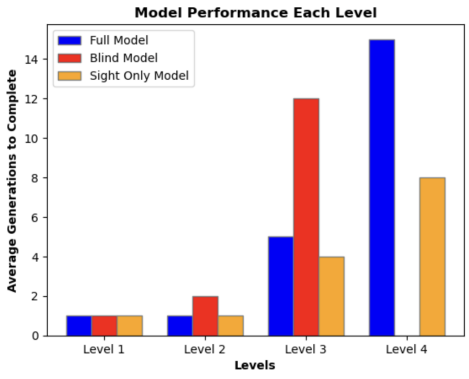

Figure 2: Comparing Each Model Measured by Median Number of Generations Until First Completion, Across All Levels

The graph above depicts the results found across all three models with the same genetic matrix functions and ten hidden layers. As shown in the data, the sight-only model performed the best overall, as well the best on each level past level one. The most basic model, the blind model, performed the worst on levels two and three and was the only model unable to complete level four. Finally, the complex model performed somewhere in the middle, being able to complete level four but not at the efficiency of the sight-only model.

These results may seem surprising but are intuitive. The complex model, having the most information to work with, is more likely to generate inconsequential mutations, compared with the bare-bones sight model. The blind model, on the other hand, is too simplistic, and shows the importance of sightlines in helping our models learn how to progress through a level, similar to the importance of a player's vision of their playing field. The sight model dominates, completing level four in half the number of generations as the complex model, as it holds important information and is able to learn optimally by generating effective random mutations at the quickest pace.

6. Conclusion

Through our final project, we learned to implement a 2D platformer video game with pygame and to apply TensorFlow machine learning models towards reinforcement learning. The implications of our final project are that, with enough processing power, machines can form their own algorithms from scratch for solving complex problems in games like chess, or in the real-world, such as those in economics or healthcare.

In reflection, a benefit of using genetic learning algorithms is that they can mimic human creativity by generating unique solutions with no prior training. Additionally, due to the use of metrics instead of specific target variables, the algorithm can be versatile and easily applied to other problems. Unfortunately, some cons of our method of training are that the model is computationally expensive and easily stumped by local maxima. Although we did not explore further due to the time constraint of this project, there are certainly ways to address these shortcomings going forward. One possible way to address the issue of computational expense would be to adapt our

code to run on a GPU, and implement parallel computing when evaluating an entire generation of models at once. In addition, to address the issue of local maxima, certain triggers could be implemented, which adjust the scoring function while the genetic algorithm is running. For example, if after several generations the models are detected to be too passive, the scoring function would be adjusted to reduce the penalty for death/increase the reward for forward progress. Ultimately, we learned a lot from implementing a video game and using genetic algorithms to solve a reinforcement learning problem, and we look forward to the transfer of our project knowledge to statistics, mathematics, and coding in our future endeavors.

7. Member Contribution

Our group consists of Michael Papagni, Stephen Yu, and Arianna Zhou. Here is a split of group member contributions:

Michael Papagni

- Implemented the graphics and controls of the video game using PyGame (Weeks 7-8)
- Made and experimented with models and mutation/propagation functions; implemented sight rays (Week 9)
- Ran and compared results of models; contributed to the presentation (Week 10)
- Added comments to code (Finals)

Stephen Yu

- Completed 5 initial game levels (Week 8)
- Made a second model; adapted the mutation and propagation functions (Week 9)
- Compiled final report (Finals)

Arianna Zhou

- Contributed with propagation/model selection functions (Week 8)
- Made a third model (Week 9)
- Made the presentation (Week 10)
- Added comments to code, compiled final report (Finals)

8. Github Link

`https://github.com/stephenyu2/pic16b-final-project`