Playing Geometry Dash with Convolutional Neural Networks

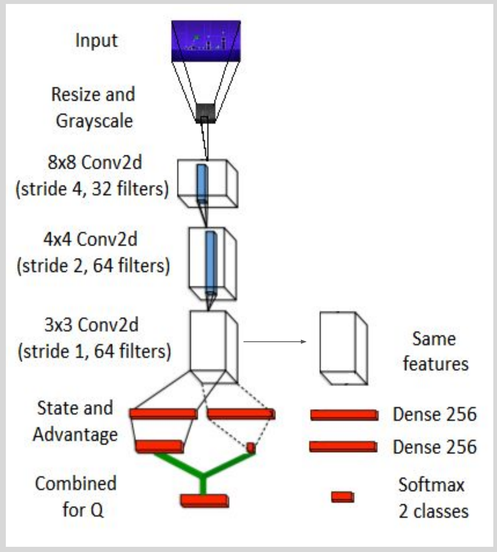
Ted Li, Sean Rafferty
Stanford CS 231n^2, Spring 2017

Introduction

Geometry dash is a rhythm based platformer that requires the player to avoid obstacles by either jumping or idling as the character moves forward through the map. In our project, we set out to create an agent that can play the game using only visual data from the game. We take two approaches: first as a reinforcement learning problem and then as a classification problem. For both approaches, we use preprocessed frames from the emulator as inputs, generate features, and then chose an action. We compare the results of these two frameworks and explore possible pros and cons of each.

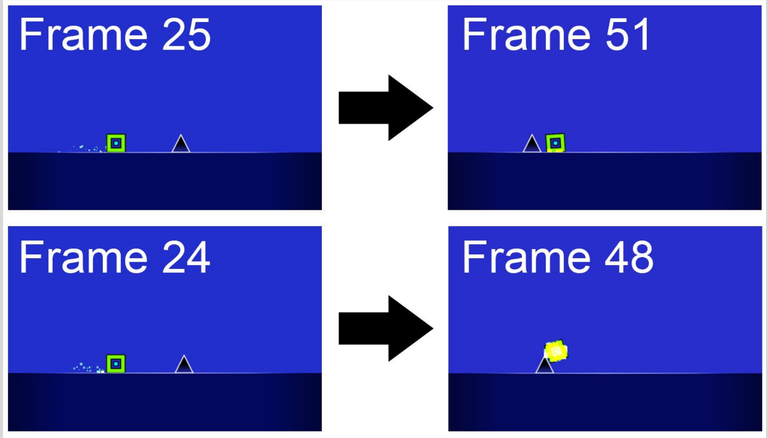
Model

Reinforcement learning Dense Classifier



Problems Encountered

Jumping just one frame early can result in failing the level, and the visual difference between these two frames is miniscule. Furthermore, the collision may occur many frames later (24 in this example). While the negative reinforcement will propagate to this action, it will also propagate to the previous frames where we did (and should not have) jumped, causing us to jump even earlier. Hence, reinforcement learning is not a good fit for our task.



Conclusion/Future Directions

We began constructing our model with reinforcement learning using a Deep-Q Network. However, upon experimentation, we learned that this seemingly simple game was far more challenging as a reinforcement learning task. Reasons for this included delayed failure and reward propagation, lack of intermediary rewards, multiple-step dependencies, and the sharp, binary nature of outcomes. While we believe a reinforcement learning model can overcome these issues, we found that training a binary classifier was more efficient and could achieve far better results with significantly less time. However, the classifier generalizes poorly to new levels. Overall, we believe that with much more data and time, both of these models would be able to successfully generalize to and complete a variety of new levels.

Emulation and Data Collection

We designed our emulator to be compatible with an OpenAI Gym environment. This required us to implement a *step* function which takes an action, uses it to emulate a step in the game, and returns the resulting state, reward, and whether that state is terminal. To make the game discrete, we injected a DLL into the application which causes all Windows API calls that return a time to instead return a fabricated time that is incremented by the inverse frame rate every step. To input actions and capture frames we used traditional Windows API calls. To detect terminal states we checked for the game over overlay. All of this functionality was then wrapped into a python library so that it could be used easily.

Experimental Evaluation and Results

We ran three simple tests to get an idea for how our model generalizes. We collected a complete run from the first two levels of the game. The second level contains a jump pad mechanic while the first level does not. Both levels contain similar obstacles. In the first four tests we train and test on the same level, splitting the half of the data both randomly and in the middle. In the last two tests we train on the entire level and test on the other level. We will explore ways to generalize learning from one level to another.

Test Name	Training Accuracy	Training % Jumps	Test Accuracy	Test % Jumps
Level 1 50/50 random split	0.988	0.445	0.928	0.442
Level 2 50/50 random split	0.978	0.465	0.872	0.444
Level 1 50/50 middle split	0.987	0.376	0.697	0.511
Level 2 50/50 middle split	0.988	0.523	0.564	0.387
Train Level 1, Test Level 2	0.985	0.444	0.665	0.443
Train Level 2, Test Level 1	0.984	0.455	0.618	0.455

Acknowledgements

We'd like to thank the CS231n instructors and TAs for teaching the class and helping with our project.

References

https://openai.com/
https://arxiv.org/pdf/1511.06581.pdf
http://www.nature.com/nature/journal/v518/n7540/abs/nature14236.html