CS230

Learn to play Candy Crush

Bowen Deng
Department of Computer Science
Stanford University
bdeng2@stanford.edu

Abstract

This work innovatively proposes to teach an AI to play the Candy Crush game. It investigated using deep Q network (DQN) to learn a policy network to help determine the optimal next step to take. Several baseline approaches are compared against the DQN agent. The code is available at [1]. Demo (with old architecture) is available at [2].

1 Introduction

Candy Crush is a very popular combinatorial game, and the major goal of the game is to gain as much score as possible within limited number of moves.
The Candy Crush game is Markovian: the distribution of the reward gained with action a at timestamp t only relies on the board state S_t, and is independent from previous states. It's also easy to represent the game state in a matrix of dimension $(C + S) \times N \times N$, where C is number of different candy colors, S is number of special candies, and N is the size of the board.
In this work, we trained a deep Q network (DQN) to learn the policy network, given the current board state.
To evaluate, we generated random board configurations and compares overall reward with arbitrary pick and Monte Carlo Tree Search as baseline, DQN with instant reward and DQN with Monte Carlo reward as experiments.

2 Related work

The company who created Candy Crush, called King, has some experiments on teaching AI to play Candy Crush [3]. But there is no open source code to benchmark, and the task is tailored to promote game design, rather than the vanilla optimization task.
There is a similar project setup, without using deep learning [4]. This work framed the problem as a graph search task, and applies standard graph search technique. This approach is weak at generalization, which is exactly the benefits of deep learning.
[5] is a work using Monte Carlo tree search to predict human success rate.
There is even a project using CNN to train an agent to learn from human play data [6], which is extremely relevant to our work. This work also used Monte Carlo Tree Search as the baseline.
More similar works are available, but none of these works uses the reinforcement learning setup we proposed. And the algorithms we proposed here is generalizable to other games and more complicated game levels, and does not require any additional training data provided by human.

CS230: Deep Learning, Winter 2020, Stanford University, CA.

3 Dataset and Features

The scope of this project is to learn an AI agent to maximize the total reward within 40 moves in a 10×10 board without any obstacles.
We implemented the game logic using PyGame module so we have full control over the reproducibility and game state and reward capturing.
At each run of the game, the program will load a configuration file containing a large fixed size of array of colors. The candy that is dropped each time is determined using the pre-determined array. Therefore, if two agents are using the same game configuration, and uses the same strategy, they will get exactly same results every time, and we are able to fairly compare different agents.
The agents are trained on training configuration file, and evaluated on evaluation configuration file, which are distinct.
In the unlikely case that the board state has appeared in the training stage, the next state will almost surely become different because it's very unlikely that the incoming colors are also identical to that of the training configuration file.

4 Methods

We proposed two different DQN agents, one using instant reward, another using Monte Carlo reward. The only difference in them is how their reward function is defined, but the algorithm are identical.
The motivation is to learn a policy neural network, which predicts the score of each action, given a game state.
To decide which action to take during training, we select the optimal action using current policy neural network with probability $1 - \varepsilon$, and pick a random action otherwise, therefore we can have exploration/exploitation trade off.
Though developed independently, our game state modeling turns out very similar to that of [6], so we borrow the image in that paper.

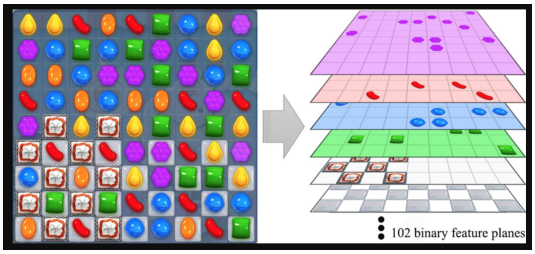

Figure 1: Game state representation

Notice that the two special layers are not within the scope of our project, and should be replaced with four types of special candies: horizontal strip bomb, vertical strip bomb, wrapped bomb, and color bomb.
There are $2 \times N \times (N-1)$ possible actions: the agent can pick a cell that is not at rightmost column and swaps with its right neighbor; or pick a cell that is not at bottommost column and swaps with its bottom neighbor.
Standard DQN objective is used, we try to minimize the squared difference between demoted target network plus reward, and the policy network. And target network is synced with policy network periodically.
With instant reward agent, r is the score obtained by picking a at state s; with Monte Carlo agent, r is the expected score obtained by picking a at state s, computed as the average of scores with random future configurations.

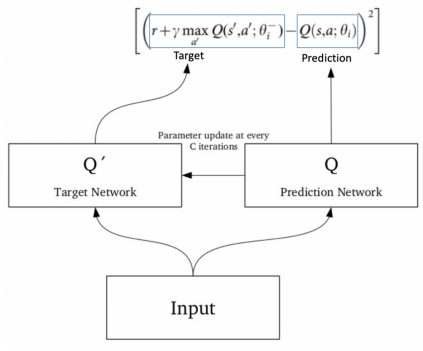

Figure 2: DQN Architecture

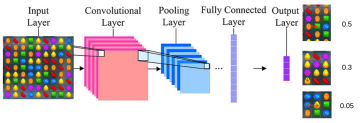

Figure 3: CNN Architecture

5 Experiments/Results/Discussion

We started with a simple architecture shown in figure 4.
With 1k evaluation configurations, the (rounded) average reward of 40 moves is as following:

Agent	Average reward
Monte Carlo tree search	325
Arbitrary pick	157
DQN with instant reward	217
DQN with Monte Carlo reward	189

From evaluation result, DQN with instant reward and Monte Carlo reward are both much better than arbitrary pick, but both much worse than Monte Carlo tree search.
By plotting the average loss per episode, we gets the loss curve as figure 5.

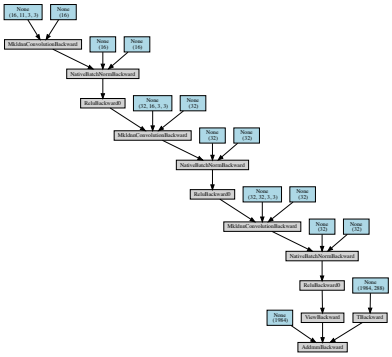

Figure 4: Old Architecture

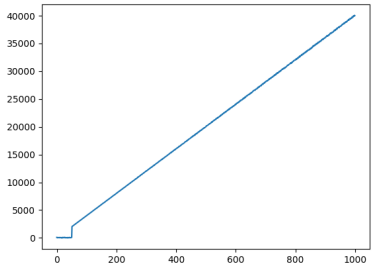

Figure 5: Average Loss with Old Architecture

As the loss suggested, the neural network is clearly underfitting the training data. Hence, more complicated network architecture/additional features are required.
A first observation is DQN with Monte Carlo reward does not make mathematical sense: the reason is that Q-learning relies on Bellman equation, which assumes the reward is due to the action a on state s, which derives new state s'. But the Monte Carlo reward is a randomized estimation of that instant reward instead. Hence, it's removed as a candidate agent.
The initial experiments was on introducing supplementary features, several options were explored:

- Created four boolean masks indicating whether this candy can be swapped with its top/left/bottom/right neighbor.

- Created one numerical mask counting number of same color candies in 8 neighbors.

It turned out that this change did not mitigate the issue.
The next attempt was to print out the gradients norm of the network, but there is no symptom on gradient being close to zero or explode.
Hence, the final experimentation focused on tuning the network architecture by applying a deeper neural network.
With dozens of experiments, we finally pick the new architecture shown in figure 6.
The final model is trained on AWS with 2k episodes.
During inference, we use the local laptop because inference does not consume as much resource as training, and it's much easier to debug.
The new evaluation table is as following:

Agent	Average reward
Monte Carlo tree search	325
Arbitrary pick	157
DQN with instant reward	370

DQN with instant reward is much better than the Monte Carlo tree search baseline, though a test against human being indicates it's a little worse than human expert.
As a proof this new architecture is getting reasonable results, apart from the favorable metrics, we also include the average loss per episodes.

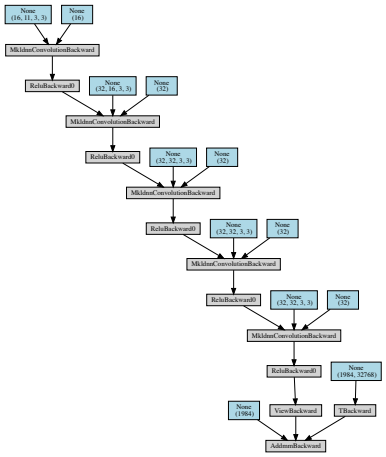

Figure 6: New Architecture

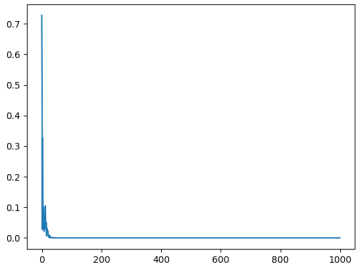

Figure 7: Average Loss with New Architecture

6 Conclusion/Future Work

After exploring several feature extraction options and experimenting with different network architectures, the final neural network is able to beat Monte Carlo tree search baseline by a reasonable amount, and with a local testing with human player, it is slightly worse than the author, but is quite close.
Due to the limited time, there are several interesting ideas that can be attempted which are not yet implemented in this project.

- Consider making the neural network permutation invariant, one simple possible way is to add a loss on permutation deviation, i.e. absolute difference between Q value for same action and two permutated board configurations.
- Implement DDPG, which can more efficiently learn the policy directly.
- Learn a shadow network which tries to predict a human expert's move.

References

[1] https://github.com/baolidakai/candy_crush_ai.

[2] https://github.com/baolidakai/candy_crush_ai/raw/master/demo.mov.

[3] https://www.gdcvault.com/play/1023858/How-King-Uses-AI-in.

[4] https://github.com/SidduSai/Candy-Crush-AI.

[5] Erik Ragnar Poromaa. Crushing candy crush: predicting human success rate in a mobile game using monte-carlo tree search, 2017.

[6] Stefan Gudmundsson, Philipp Eisen, Erik Poromaa, Alex Nodet, Sami Purmonen, Bartlomiej Kozakowski, Richard Meurling, and Lele Cao. Human-like playtesting with deep learning. pages 1–8, 08 2018.