CandyCrush.ai: An AI Agent for Candy Crush

Jiwoo Lee, Niranjan Balachandar, Karan Singhal

December 16, 2016

1 Introduction

Candy Crush, a mobile puzzle game, has become very popular in the past few years. The game looks deceptively simple, but it turns out to be a nontrivial task to "solve" Candy Crush. Though there are random elements to the game, there are valid strategies to employ in order to maximize the score. The state space is extremely large (on the order of 10^{50}, and getting an optimal score is an NP-hard problem [2].

In this project, we are implementing an AI agent that plays a slightly simplified variant (as defined in the Game Specification Section) of Candy Crush with a goal of getting an optimal score. We tested different algorithms using Q-Learning and function approximation and got fairly promising results.

2 Game Background

Candy Crush, like many games, is played on a grid. Each coordinate on the grid holds a candy, of which there are usually five types (usually distinguished by shape and color). Players earn points by swapping the positions of two candies to create rows or columns of 3 or more of the same candy. This is considered to be one move, and each move can obtain a score based on how many candies were connected in a row.

Any move/swap that does not result in a row or column of 3 or more of the same candy does not count as a valid move. If there are no possible moves to be made, the game shuffles its board until a move is available. Once the candies are aligned in a row or column, they disappear, and all the candies above it are shifted down (new ones refill the grid from the top of the board). If new ones also form sequences of 3 or more candies, these sequences also disappear, further adding to the score, and this continues.

The game usually has a different end condition for each level, but we will focus on maximizing the score in a limited amount of moves.

3 Previous Approaches

There are a couple Candy Crush bots online, and one is quite well documented. The Candy Crush Bot by Alexandru Ene is written in Python, and implements a greedy algorithm that performs the highest scoring move on the board at each turn. While our approach takes into consideration the highest scoring move on the board, we extrapolate more from the other candies on the board to take into account resulting combos to make the most optimal move [2]. Ene's algorithm fails to consider anything besides the immediately relevant candies, so our algorithm should be expected to perform significantly better.

That being said, we are not exactly sure how Candy Crush replaces the necessary candies after a row or column is deleted. It is likely that Candy Crush has implemented some minimax heuristic to prevent things like a combo of 13 (an extreme outlier we found in one of our trial runs) which would increase the score exponentially. Unfortunately we are unable to compare the exact scores between the two bots as his Candy Crush bot plays the full version of Candy Crush while we have modeled a simpler version of the game. We are confident however that we would fare well against Ene's bot provided that we also put more thought into the minimax algorithm that Candy Crush might be using.

4 Approach

4.1 Game Specification

We chose to use an MDP to represent the game because an MDP involves transition probabilities over possible states given initial states and actions. Given that the outcome of a move in Candy Crush is not fixed for a given initial state and there is a degree of randomness, transitions allow us to describe the large state space of the game probabilistically, even if we do not manually specify every transition.

We specify the game's MDP as follows:

States: States are specified as pairs of current board game states, represented as a 2D grid of numbers on the board, and number of turns left.

Start State: The start state is a randomly generated grid of numbers and NTURNS turns left, where NTURNS is some constant number of turns per game (we use 50).

Actions: The set of actions from a given state is the set of valid pairs of coordinates that can be swapped during a move such that the swap results in a row or column of 3 adjacent matching numbers.

Transition Probability: We do not fully specify a transition model, nor do we attempt to learn it in our approach, as the state space is too large to specify every transition between game states, and our approach is not model-based. The state space ends up being huge because one combo could lead to more combos being triggered which increases the state space exponentially.

Rewards: The reward for any given action and state is what define as the turnScore function in our proposal. As we've outlined earlier, the turnScore function at combo number i, if the biggest group of connected candies in a row/column is of size c, where $c \geq 3$, returns the score for turn t, combo i as:

$$\text{turnScore}_i(c) = (10c^2 - 10c)i$$

We have set this scoring function to mimic the scoring of the mobile Candy Crush application. This scoring function rewards swaps that result in combos (when a swap results in multiple rows/columns of at least 3). The total score for a turn is the sum of the turn scores for all iterations i performed for that turn:

$$\text{turnScore}(t) = \sum_{\text{all } i} \text{turnScore}_i$$

End state: An end state is any state in which the number of turns is 0.

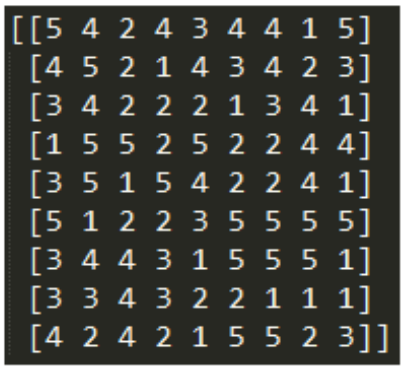
We show two states of the game board, including one end state in Fig. 1

4.2 Baseline

Our baseline was choosing the first valid move the agent encounters. We chose this as our baseline because it is the absolute lowest baseline that any Candy Crush agent should beat. Any more advanced bot must at least beat this baseline.

4.3 Oracle (Human/Greedy)

Our oracle was initially human gameplay, but we found that even our initial approach quickly outperformed human gameplay, so we implemented a greedy approach (as specified by Ene) that chooses the best possible move at each turn (the move with the greatest guaranteed increase in score) [2].



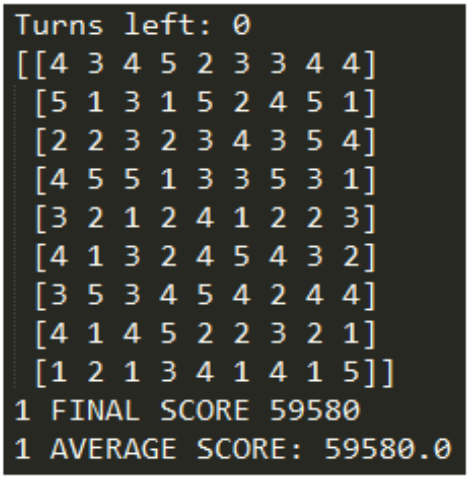


Figure 1: Two game states in our specified MDP. Each iteration of the game calculates a final score and updates the average score.

4.4 Model-Free Approach Motivation

Given that the game is represented as an MDP, the challenge is for an advanced bot to understand the MDP well enough that it can make good moves given any state and action. If the model can predict what the best action is given a state, it can do as well as possible, given the probabilistic nature of the game.

There are two paths forward here: we can use a model-based approach, in which we attempt to learn the transition and reward model and use it to evaluate the value of different states and actions, or we can use a model-free approach, in which we directly learn values of states and actions.

There is a trade-off here between accuracy and computational complexity, as a model-based approach would likely be more robust, especially for unseen states, but a model-free approach would likely be more computationally feasible. We chose the latter for Candy Crush, as the size of the state space and the resulting complexity of any predictive transition and reward models makes learning them difficult.

4.5 Q-Learning with On-the-Fly Game Simulation

4.5.1 Motivation

A model-free approach also introduces the possibility of learning values on the fly, which is key to our initial approach. Instead of combining complete transition and reward models to produce Q values for every state and action, we can selectively compute Q values for states and actions relevant to the current move decision as the bot makes moves. This significantly reduces the computation necessary for the bot to make good moves.

Then, as a move is being made, this approach requires the bot to generate data about the current state that would allow it to update Q values for the state and possible actions from it. Two algorithms that allows us to do this are Q-learning and SARSA.

We chose Q-learning because Q values are more likely to converge to optimal values given an exploration policy and the limited number of samples we can generate on the fly.

To generate data about the current state, we chose to generate a constant number of episodes ending in an end state. We did this to ensure our Q values were not short-sighted and so that future states would have meaningfully initialized Q values to inform exploration.

4.5.2 Details of Approach

This approach has no separate training and evaluation phases. We found that saving Q values from previous runs had no effect on score, but it increased run time, possibly due to the blowup in storage requirements associated with storing values for every state-action pair previously encountered. We believe that the reason saving Q values from previous runs did not increase score was that every state-action pair was exceedingly unlikely to be encountered twice, even if many games are played.

As a result, we found no need for a training phase. Rather, as the game is being played, for each move, we did 25 Monte Carlo simulations until the end of the game using an epsilon-greedy exploration policy.

More precisely, during exploration, in the case that Q values for an initial state is not initialized, then a random action is chosen for the episode. If Q is initialized, then we use an epsilon-greedy exploration strategy with $\epsilon = .25$. This was to ensure the samples were diverse enough to produce updates allowing Q to converge to optimal values.

Q-values of the current state-action pair are updated to be the average final score from these simulations. This is repeated for each move, until there are no more moves to be made.

4.6 Q-Learning with Linear Function Approximation

4.6.1 Motivation

We found that Q-Learning with on-the-fly Game Simulation wasted computation, as knowledge learned about Q values for specific state-action pairs should be reflected in similar state-action pairs. In other words, we needed to generalize our Q-Learning.

This is especially important for a problem with such a large state space, since state-action pairs are virtually never encountered again. Generalizing also introduces the possibility of discrete training and evaluation phases, as knowledge learned from previous games are now useful for future games even if those future games do not involve repeated state-action pairs.

4.6.2 Details of Approach

We initialize weights for our linear function to be 0 before training.

During the training phase, we repeated the on-the-fly simulation of the previous approach, and we recorded each simulation as a series of (state, action, reward, newState), or (s, a, r, s'), tuples, where the reward is specified as the change in score.

To minimize the squared loss function,

$$\min_{w} \sum_{(s,a,r,s')} (Q_{Opt}(s,a;w) - (r + \gamma \max_{a'} Q_{Opt}(s',a';w)))^2$$

where $Q_{Opt}(s,a;w) = \phi(s,a) \cdot w$ and $\gamma = .5$. We then performed the following update for every tuple in the recorded series for every simulation. After each turn we had a state (s), chose an action (a), obtained a reward (r), and produced a new state (s'). On each (s,a,r,s'):

$$w \leftarrow w - \eta[\hat{Q}(s,a;w) - (r + \gamma \hat{V}(s'))]\phi(s,a) \tag{1}$$

where $\eta = 0.00000000001$ where ϕ is the feature extractor defined as follows:

Feature Extractor

For each (state, action) pair (s,a), we used the following 24 features to calculate $\phi(s,a)$.

1. As a reminder, each action is a pair of coordinates representing candies that are to be switched. This feature is the minimum of the two coordinate rows. The rationale is it is possible that switches towards the bottom of the board are favored over switches towards the top of the board and vice versa.

2. This feature is the maximum of the two coordinate rows. The rationale is the same as that for feature 1, plus it is possible that actions where the max row is greater than the min row are favored over actions where the max row is equal to the min row, or vice versa.

3. This feature is an identity feature vector for whether or not the action represents a switch between a pair of coordinates in the same column. The rationale is it might be more favorable to make switches within the same column than it is to make switches in the same row or vice versa.

4. This feature is the number of valid moves (i.e. number of pairs of adjacent candies with

the same color). The rationale is states with more valid moves might be more likely to yield larger deletions and more combos and chain reactions of deletions, producing a higher score.

5. This feature is the median utility (discount factor $\gamma = 0.5$) of 25 simulated episodes starting with the current state and running an episode until a limited depth of 5 new states or until an end state. The first action is the input action to the feature extractor, and the subsequent actions are chosen randomly from the set of available actions for the current state. The rationale is similar to that for generating SARS' episodes in regular Q-learning; by sampling episodes, we can estimate the value of a state and action. Episodes are limited in depth to reduce run time, and the median utilty is calculated to avoid skewing produced by episodes where, by random chance, there were many combos and the utility was unusually high.

6. This feature is the number of candies that will be deleted immediately after a switch. These feature does not consider future deletions, combos, and chain deletions. The rationale is while feature 5 estimates the value of a state, action pair, this would give a concrete number for the immediate number of candies guaranteed to be deleted.

7-15. These 9 features are the maximum count of any one candy in each of the rows 1-9. For instance, if row 8 contained the following candy types (represented by numbers): [1,2,3,4,3,4,4,5,2], feature 8 would be 3 because the most common candy type is type 4, which has a count of 3. The rationale is if one type of candy dominates a row or column, it is more likely a large number of candies in a row or can be deleted or combos can occur in the row from a deletion elsewhere.

16-24. These 9 features are the maximum count of any one candy in each of the columns 1-9. The rational is the same as that for features 7-15.

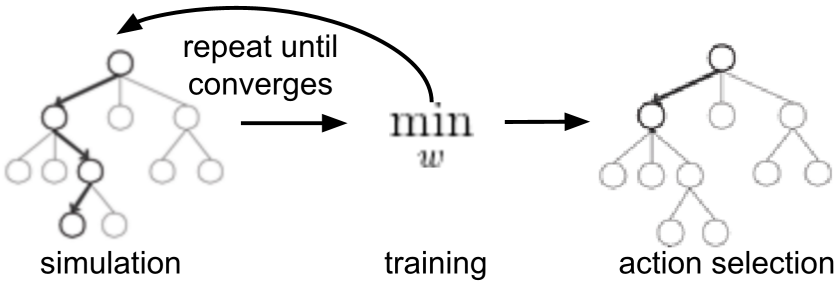


Figure 2: Our training approach for Q-Learning with Linear Function Approximation is summarized by this workflow.

Our training approach Q-Learning with Linear Function Approximation is summarized in Fig. 2. During training we ran games where for each turn, we chose a policy with an epsilon-greedy approach ($\epsilon = 0.5$, i.e. half the time we chose the optimal policy and half the time we chose a random policy), and updated weights according to Equation (1). We trained the learner until weights converged.

During testing, we used the converged weights from training as a constant weight vector to chose an optimal action at each turn.

4.7 Q-Learning with Neural Network

4.7.1 Motivation

After implementing Q-Learning with Linear Function Approximation, we wondered if we could extend the hypothesis class for our function approximation by experimenting with a neural network. We hoped that a neural network could provide a more expressive function that better captures the interactions between and the non-regularities and non-linearity in the features given by our feature extractor.

4.7.2 Details of Approach

Our neural network is used to predict Q values given $\phi(s, a)$. It is a simple multi-layer perceptron with three layers. Its input layer has 25 neurons, one for each feature in the feature extractor used for linear approximation (the neural network uses the same feature extractor detailed in **4.6.2**) and one additional neuron with an input value of "1" to serve as a bias. Since the neural network is a regressor, it has one neuron in its output later. It has a single hidden layer with 11 neurons, including one bias neuron (Note that in our code, the bias neurons in both the input and hidden layer are not included in the network's counts of neurons for these layers, and their weights are referred to as "intercepts" instead of "coefficients", unlike the rest of the weights). The activation function for the hidden layer is ReLU, or the Rectified Linear Unit function, which takes the max of the input and 0. The activation function for the output layer is just the identity function.

The neural network minimizes over the same loss function as the linear function approximation. It does this using Adam, a modified gradient descent algorithm proposed in 2015 that performs well on certain datasets [3].

Again, during training, we ran games where for each turn, we chose a policy with an epsilon-greedy approach ($\epsilon = 0.5$). The neural net partially fits each data point ($\phi(s, a)$, target) on each (s, a, r, s') tuple during the training phase, replacing the weight update step for the linear approximation with the partial fitting step. During partial fitting, the neural network is trained to minimize training loss on data values partially fitted so far. Losses are backpropogated to update weights between the hidden and output and input and hidden layers. This is repeated for a maximum of 200 iterations or until convergence.

During testing, we used the converged neural network weights from training to specify a neural network used for prediction. We used this neural network to chose an an optimal action at every turn by predicting the Q values of every state-action pair from a state and taking the action that produces the maximum value.

5 Results and Discussion

The following results are all for 50-turn games. We ran 10 games with human, 1,000 for each of baseline and greedy, and 100 for each of regular Q-learning, Q-learning with linear function approximation, and Q-learning with Neural Network. Fig. 3 shows the average score across all games for each of these algorithms.

Human

The human player was worse than each of the algorithms, except for the baseline. On average, the human took about 8 seconds per turn. Because it is very hard for the human to analyze the entire game grid within a reasonable time (< 15 seconds), the human player focused on making switches towards the bottom of the grid, so chain reactions and combos are more likely to occur.

Greedy (Oracle)

The greedy algorithm performed better than the human. This is likely because the greedy algorithm is able to very quickly examine all possible valid actions at each turn, while the human was limited to certain regions of the board due to make moves in reasonable time. On average, the greedy algorithm took much less than 1 second per turn.

Q-Learning with On-the-Fly Sampling

In our trials, we set the training $\epsilon = 0.5$, step size (η) to 0.2, our discount (γ) to 0.5, the number of SARS' episodes to generate at each turn) to 25. The more SARS' episodes we generate, the more accurately we can estimate the value of a state. However, there is a trade-off between the number of episodes we generate and the run time, so we chose 25. We chose Each turn took about 3 seconds, so each iteration (full game) took about 2 to 3 minutes to finish, so running these 66 trials took about 3 hours.

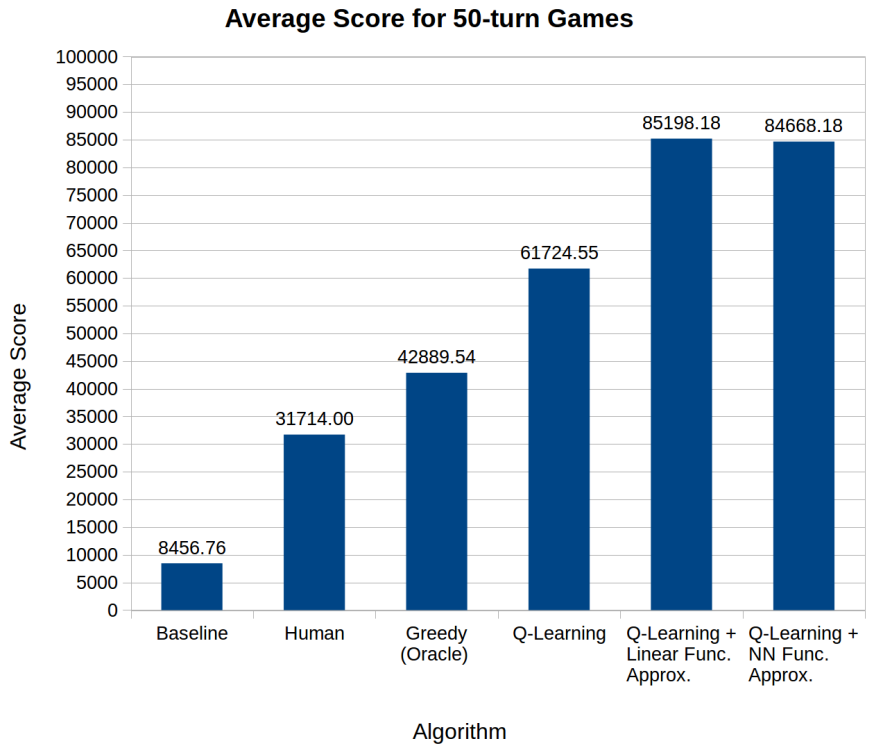


Figure 3: Q-learning results: Average score for each algorithm for 50-turn games. As expected, the baseline performs the worst, while Q-learning with function approximation performs the best.

Q-Learning with Linear Function Approximation

For feature extraction, we set discount (γ) to 0.5 and the number of SARS' episodes to generate when calculating $\phi(s, a)$) to 25 with a limited depth of 5. The more SARS' episodes we generate and greater the depth of these episodes, the better we can estimate Q_{opt}. Again, there is a trade-off between the number of episodes and run time, so we chose 25 with depth limited to 5 because these were already yielding much better results than regular Q-learning. For training, we set the exploration probability $\epsilon = 0.5$, and step size (η) to 0.00000000001, Each turn took about 5 seconds, so each iteration (full game) took about 4 to 5 minutes to finish, so running these 66 trials took about 5 hours.

Linear Function approximation performed the best out of all the algorithms (about equal to Neural Network). With good features, function approximation was expected to perform better than regular Q-learning because instead of relying on sampling on the fly, the linear function weights reflect several hundreds (or thousands) of iterations of game-playing and exploration during training.

Q-Learning with Neural Network

We used the same feature extraction and training constants that we used for linear function approximation. Again, each turn took about 5 seconds, so each iteration (full game) took about 4 to 5 minutes to finish, so running these 66 trials took about 5 hours. The neural network performed about equally to the linear function approximation algorithm.

Because expanding the hypothesis class did not appear to increase performance significantly, it appears that (given our features) the best-performing model is linear or close to linear. If we take into account Occam's Razor, this would suggest Q-Learning with Linear Function Approximation is a better model, as it is simpler to understand and achieves about the same results.

Another possible explanation is that the best function given our features cannot be expressed by a 3 layer neural network and a deeper network would perform better, but this is unlikely given that there was no significant increase in score with the addition of a single hidden layer.

6 Future Work

6.1 Optimization

In each of our algorithms, we feel that there is some room for improvement. We have not optimized the constants our algorithms use, including the discount for future rewards, step size for updates to Q values, number of episodes generated to produce samples in Q-Learning, and our eta in our function approximation updates. We have also not experimented with exploration strategy, and we believe softmax may produce samples that better balance exploration and exploitation than epsilon-greedy. Unfortunately we couldn't formulate a strategy to optimize the constants beyond just running time-consuming trials and picking values that yielded the greatest score, so we put the majority of our effort into experimenting with algorithms.

6.2 Game Generalization

In our model of the game and consequently our algorithms, we play a simplified version of the Candy Crush where the grid is an unobstructed 9 by 9 grid of 5 pieces. Though the actual version of Candy Crush starts this simple, as the game progresses, more and more types of candies are added and obstacles are placed within the board. The objectives of the levels can also vary from needing to score above a certain threshold in a limited number of moves/time to trying to move certain blocks to the bottom of the board. Unfortunately our algorithms are currently not general enough to, say, play a game that requires that 100 red pieces are deleted; as this game would have a win or lose outcome, we would need to change our scoring function accordingly. A possible modification to our algorithm would be flexibility in handling any type of objective on any type of map. Moving more in the direction of general game playing, however, may lose some of the benefits of domain knowledge of strategy for particular games.

However, there are advantages to our current generalized algorithms. We found out halfway into the project that we modeled a game more similar to Bejeweled than Candy Crush, so our algorithm would work perfectly in that game also. As it turns out, many grid based games that require moving and deleting pieces exist; a modification to our algorithm could perhaps generalize our strategy to many grid based games that require an optimal score.

References

[1] Walsh, T. Candy Crush is NP-Hard. *NICTA and University of NSW, Sydney, Australia.* https://arxiv.org/pdf/1403.1911v1.pdf
[2] Alex Ene's Candy Crush Bot. https://github.com/AlexEne/CCrush-Bot
[3] Ba, J., Kingma, D. ADAM: A METHOD FOR STOCHASTIC OPTIMIZATION. *ICLR 2015.* https://arxiv.org/pdf/1412.6980.pdf