Playing "Dominion" with Deep Reinforcement Learning

Garrick Fernandez
Department of Computer Science
Stanford University
Stanford, CA
garrick@cs.stanford.edu

Benjamin Anderson
Department of Computer Science
Stanford University
Stanford, CA
banders9@stanford.edu

***Abstract*—Developing a good strategy in a multi-player game requires dealing with a significant amount of uncertainty, due to the unknown behavior of other agents, as well as inherent randomness. In our project, we apply deep reinforcement learning to the multi-player card game "Dominion." Our RL agent was able to beat a computer player which selected a random action every turn. Subjectively (judging by our domain knowledge), the agent seemed to often choose fairly reasonable actions, though not optimal. With more self-play and more sophisticated feature extraction, we hypothesize that our agent could get even better.**

"You are a monarch, like your parents before you, a ruler of a small pleasant kingdom of rivers and evergreens. Unlike your parents, however, you have hopes and dreams! You want a bigger and more pleasant kingdom, with more rivers and a wider variety of trees. You want a Dominion! But wait! Several other monarchs have had the exact same idea. You must race to get as much of the unclaimed land as possible, fending them off along the way." – Dominion game box

I. Introduction

Reinforcement learning (RL) describes a set of approaches for making decisions in sequential problems (Markov decision processes) where the consequences of these choices are unknown. Just as an animal can learn behavior based on rewards and punishments (e.g. treats or electric shocks), reinforcement learning algorithms such as Q-Learning (Watkins, 1989) [1] and SARSA (Rummery & Niranjan, 1994) [2] leverage dynamic programming to allow an artificial agent to learn an optimal 'policy' from experience.

One interesting domain where agents must a make a sequence of decisions under conditions of uncertainty is game-playing. In a zero-sum multi-player game, players take turns making decisions, and the conditions of the game evolve based on those decisions, until someone wins the game, which is the "reward." From the viewpoint of a single player, each of her decisions lead to an outcome that she cannot be sure of, because it depends on what other players do, and sometimes on randomness built into the game. Thus, although there search algorithms designed specifically for games, it can also make sense to treat some games as Markov decision processes, and learn a strategy with reinforcement learning. We review several successful applications of reinforcement learning to games in Section II.

In this paper, we explore the application of reinforcement learning to the game of Dominion. Dominion is a two- to four-player "deck-building" game, so called because each player begins the game with a small, weak deck of cards, and over the course of the game, purchases more cards to add to his or her deck to improve it. At each turn, players must make decisions about what cards to play and what cards to purchase. These decisions occur under conditions of uncertainty due principally to (a) the random order of cards in their deck upon shuffling; and (b) the choices made by other players.

Because the state space of the game is large, effective learning requires generalization about the expected value of taking each action from a given state. We use deep learning (i.e. neural networks) to approximate the value of unseen states and actions from those that have been observed, and use the SARSA algorithm to update the agent's beliefs about these values from experience.

With this approach, we were able to teach our agent to play Dominion well enough to beat a naive "random" policy. We also noticed significant improvement in estimating Q-values over repeated iterations of self-play.

II. Related Work

A. Reinforcement Learning and Deep RL

Reinforcement learning operates in an environment where transitions and rewards are unknown. Given this uncertainty, one can either try to estimate transitions probabilities and rewards (model-based RL), or directly estimate the value of taking some action from some state (model-free RL). One of the earliest model-free approaches is Q-Learning, which uses bootstrapping (estimates based on previous estimates) to determine the "Q-value" of taking action a from state s, according to the following equation [3]:

$$Q(s,a) \leftarrow Q(s,a) + \alpha\Big(r + \gamma\, max_{a'}\, Q(s',a') - Q(s,a)\Big)$$

This equation 're-estimates' the value of $Q(s,a)$ as the reward earned, plus the estimated Q-value of the subsequent state (guessing the "best" action from this state), discounted by γ. Then, $Q(s,a)$ is updated as a weighted average of the new and old estimate, where the learning rate α determines the weight of the new estimate. SARSA is similar to Q-Learning, except

that it requires a pre-determined policy, and estimates the Q-value based on the next action taken by this policy, rather than maximizing over all possible actions. With suitable policies to encourage exploration, both of these algorithms converge to the true Q-values. [3]

Q-Learning and SARSA as originally conceived require a "lookup table" for the Q-value of a state-action pair. [2] In environments like Dominion (and other complex games), where the state space is enormous or infinite, it is not possible to even experience every state during learning, let alone store all of these Q-values. Instead, we must *generalize:* infer the Q-values of states we haven't seen from those we have, which requires extracting the relevant features of a state, and determining how those features relate to the value of taking each action from that state.

One approach for doing so is *deep learning*, which leverages neural networks to learn a predictor that can be non-linear and very expressive (as each layer of the neural network can represent a higher level of abstraction based on the previous layer). For example, it can learn abstract features of visual input expressed in pixels [4]. The use of deep learning to solve this generalization challenge in reinforcement learning is known as *deep reinforcement learning*, and it has been responsible for many of the successful applications of reinforcement learning to challenging, real-world problems, from robotics to finance [4]. It has also been applied successfully in game-playing, and some of this work will be discussed in the following section.

B. Game-Playing with RL

As mentioned in the introduction, many games can reasonably be treated as sequential decision processes with uncertain transition functions and rewards. Reinforcement learning can then be applied to develop a strategy or policy for playing the game, which allows an AI agent to play autonomously. Mnih et al. used Deep RL to successfully play a wide many Atari games at or above a human level of proficiency simply by observing pixel values as the "state," and using controller inputs as actions. [5] Deep RL has been successfully applied to multi-player games as well: for example, Silver et al. used neural networks and reinforcement learning to develop an agent for the (incredibly complex) board game Go, which played just as well as "state-of-the art" Monte Carlo Tree Search programs. [6]

Previous *Decision Making Under Uncertainty* student projects have also applied reinforcement learning to adversarial games, such as Texas Hold 'Em [7] and *Super Smash Bros. Melee* [8]. We were particularly interested in the Smash project because it successfully applied a perceptron to generalize across a vast state space. Deep neural networks are a more complicated approach, but they share the basic framework of perceptrons, which is taking in a set of inputs and using a learned function to output a prediction for the value of a state-action pair.

III. Problem and Approach

A. The Game of Dominion

Dominion is a multiplayer, adversarial card game, played with a specialized set of cards. It is part of a set of games known as "deck-building" games, so called because each player begins the game with a small, weak deck of cards, and over the course of the game, uses the cards they already has to purchase more cards, which are permanently added to add to her deck to improve it.

There are three main types of cards in Dominion, shown in Figure 1 below. Treasure cards are give the player money to buy new cards. Victory cards do not do anything until the end of the game, when they are counted up to determine the winner. Action cards do a wide variety of things, including allowing players to gain cards or attack other players. (To avoid confusion, we will use "Actions" to refer to these cards, and use "actions" to refer to any decisions made by players in a sequential decision process.) At any given time, each player has a hand (which is visible to only them), a deck (which no one can see), and a face-up discard pile. The game begins with a shared "Supply" of card piles in the middle, from which players can purchase cards. Each pile has a multiplicity of eight or more, so more than one player can get copies of the same card. Some of these card types (standard Victory and Treasure cards) are in every game, while others (including Actions) are randomly selected from a larger pool of possible cards before each game, making every game of Dominion unique.

Fig. 1. Example Treasure, Victory, and Action

Each player starts the game with a small deck consisting only of Estates (the least valuable Victory card), and Coppers (the least valuable Treasure card). A turn consists of an "Action" phase, during which the player can play Actions; a "Buy" phase, during which a player can buy new cards to add to her discard pile; and a "cleanup" phase, upon which her hand, and all actions and treasures that were played, are discarded, and a new hand is drawn from the deck. When the deck runs out, the player shuffles her discard pile and it becomes her deck. In this way, because cards that are used are recycled rather than thrown away, a player's deck can grow better and better over time, as they use the cards they already have to buy more and better cards. The game ends when all Provinces (the most valuable victory card) have been bought, or when three other piles in the Supply are emptied.

The player with the most Victory points wins the game. An even more detailed description of the rules can be found on the Dominion website.

B. Game Engine

In order to conduct simulations with computer players, allowing them to play against each other and against people, we constructed a Dominion engine in Python.

C. Modeling Dominion

Whenever it is her turn, a Dominion player must make decisions about which cards to play in her Action phase, (and sometimes, additional choices as required by these Actions), and what cards to purchase during her buy phase. All of these decisions constitute "actions" in the broader game-playing sense of the term. These actions occur under conditions of uncertainty due principally to (a) the random order of cards in their deck upon shuffling; and (b) the unpredictable choices made by other players. The full state of a game of Dominion can be described by:

- The cards remaining in the Supply.
- Each player's hand, deck, and discard pile.
- Whose turn it is, what phase of their turn it is, and what they've done so far this turn (i.e. extra spending power or buys earned by Actions they have already played).

Given that in any game of Dominion, the Supply has upwards of 20 card types with multiplicity between 8 and 60, the possible permutations of hands, decks, discards, and cards in play is unimaginably large. The fact that some cards in the Supply are chosen randomly to be included before the game starts just increases this state space any more. This is what makes Dominion such a fun and novel game for humans to play, but it also explains why our agent's ability to generalize will be crucial.

We model Dominion as a Markov decision process, where the state is described by a set of relevant features that an actual person playing the game would have access to, namely her own hand, the cards in the Supply, the Victory points collected by each player, [insert more features here]. Of course, these features do not specify the *complete* game state, but our conjecture is that they provide enough information to learn what choices and what states are better or worse, and to generalize between similar-but-not-identical states.

Transitions between states depend on the behavior of other players, so they are not truly "random" (unless the agent is playing against a player who makes decisions randomly). However, our choice to model this problem as a Markov decision process means that we are effectively treating the decisions of other players as stochastic. Though this might mean we do not learn a perfect strategy, it is reasonable to think we could learn a pretty good one, especially given that there are only moderate amounts of interaction between players (players share a Supply and some Actions attack other players, but each player is otherwise building up her deck isolation).

The primary objective of a Dominion game, like most adversarial games, is to win. Thus, the principal "reward" is given to an agent if it wins the game (+100). Winning by *more* is generally considered good, so the reward is larger if you trounce your opponents (+10 $\times$ margin of victory). Finally, we include some negative rewards for behavior that is obviously idiotic, such as not buying anything when the agent has a lot of money. We also penalize players for games that take an unreasonable number of turns (hundreds and hundreds), because if everyone behaves reasonably, games should not take this long, and it is bad for training, as well as in the real world, for games of Dominion to drag on for hours. (We recognize that part of the value of AI agents is learning things human intuition might not pick up on, but some guardrails are necessary for learning to happen at a reasonable pace and achieve helpful results.)

D. Problems with Perceptron

To perform the learning task, we initially attempted SARSA with global approximation, using a perceptron to generalize to unseen states, using features from the observable game state, as described above. However, this turned out to be intractable for two reasons:

- (1) There are many possible distinct actions: during the Action phase, a player can play any card from her hand that is an Action; and during the Buy phase, a player can buy any card she can afford. Playing an Action can also trigger "sub-actions" (e.g. selecting which card to gain or which card to trash). Because a perceptron has a different set of weights for each action, attempting to use it was a nightmare because actions are so numerous and varied.
- (2) The engine is built such that the "actions" a player actually has access to are not cards, but keystrokes. (For example, to buy a Silver, one would press 3 during the Buy phase.) This means that playing Dominion with our engine is much like playing Atari: the same input can mean vastly different things in different contexts.

Because of these difficulties, we decided to use deep reinforcement learning instead. The advantage of this is that a neural network can "learn" more complicated, abstract features from granular, basic inputs. Instead of us having to somehow communicate that inputting 3 will buy a Silver in such-and-such a context, we push this into the learning task instead.

E. Deep Learning Architecture

1) Basic Idea: Our final learner was based on SARSA using a neural network for global approximation. In this scheme, input features are passed through layers of successive neurons, each of which compute a linear combination of the previous inputs and applies a nonlinear activation function to the result. At a basic level, this is similar to stacking perceptrons on top of one another. The advantage of this, as described before, is that a neural network can infer more abstract things from basic input features (for example, that pressing 1 during the Buy phase will buy a Copper). means buying a Copper, for example). The network accepts an input

vector, which encodes various features about the state and action. In contrast to the perceptron approach, we do not need different weights for each action: we push the task of varying weights depending on the action into the neural network.

We designed a content-associative tower architecture inspired by the way human players may group and process information. If a player sees information about their hand, they may derive some high-level strategy from it. This is then combined with their intuitions about their deck or the state of the game, which contribute to a final decision about what action to take. As a result, our input features are segmented and fed to several "towers" which process the content and derive a latent representation of it. These representations are concatenated and fed through more dense layers to derive the Q-Value. This is visualized in Figure 2 below.

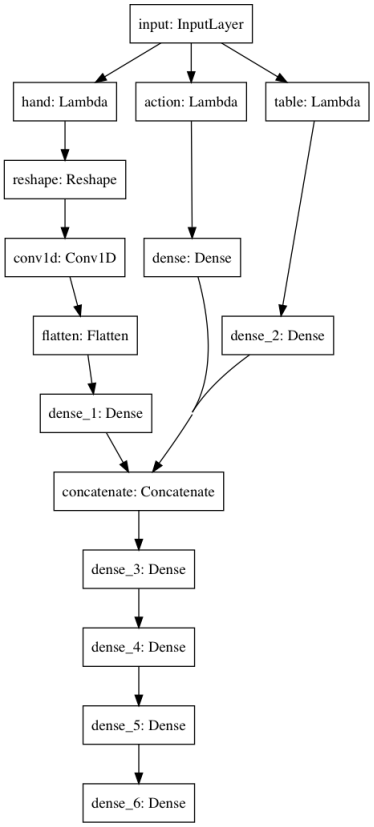

Fig. 2. Deep Learning Architecture

2) More Technical Notes: Our network was relatively small (∼12,000 parameters), relying on the content-associative approach rather than a "brute-force" approach with more layers and neurons, which eased the training time. The "hand" features are one-hot vectors encoding up to 15 cards in the player?s hand. The respective tower uses a 1D convolution to apply a filter the size of the number of cards in the game to each ', allowing for parameters to be shared. These filters can be thought of as mini-evaluations of different aspects of fitness and utility of cards in hand The initial layers used a ReLU activation (Rectified Leaky Unit), while later layers used a sigmoidal activation. The final layer consisted of one neuron with a linear activation (to predict Q-values). The model was built in Keras and trained using Adam optimization (a more stable optimizer than gradient descent) to minimize mean squared error. Rather than a traditional SARSA update with a learning rate, we simply re-estimate $\hat{Q}(s,a) = r + \gamma Q(s',a')$, and 'tell' the model that this is the correct answer, and rely on it to update the weights accordingly.

While the neural network architecture helped with the intractable action space problem, it suffered due to sparse reward. Games consist of hundreds of decisions, and the reward for most of them is zero. A neural network can cheat and guess zero most of the time and do pretty well. To help this, we introduced eligibility traces to assign credit from the end-game reward to past decisions. To implement this, we only trained the neural network after an entire game, and we distributed the final reward (with decay) back through the rewards vector.

IV. Experiments and Results

A. Overview of Experiments

As a baseline, we created an computer player ("Random") which does no learning whatsoever, and selects a random action. Then we conducted a few experiments that involved our RL agent ("Learner") playing games against Random, against other Learners (which were first trained against Random), and iteratively against itself.

B. Learner vs. Random

After training our Learner only one game against a Random opponent, it was able to beat a Random opponent in 100% of games afterwards. In this experiment, the Learner appeared to learn a policy of buying mostly Silvers and Provinces, which is a simple but fairly good strategy (it would have been even better if it learned to buy Gold). What we took away from this is that our Learner would probably need to play against a better player than Random to get very good at the game, since beating the baseline was so trivial that it only took one game of training.

C. Genetic-Style "Bracket" of Learners

Our next idea was to first train a bunch of Learners vs. a Random opponent, and then use a bracket to play those Learners against one another, resulting in a sort of "survival of the fittest," in which we would expect the winner of the tournament to be better than just an agent trained against a Random opponent. The problem with this is that as you scale up the number of levels of the bracket, the number of agents required grows exponentially, and training is quite time-consuming (it takes several minutes to play a game and learn from it). This means, for example, that to have a "level-5" agent, even training on only one game each time, we would need 16 games vs. Random, then 8 games between "level-1" agents, 4 games between "level-2" agents, 2 games between "level-3" agents, and finally a face-off between two "level-4" agents. This blows up quickly, and we found that the

agents weren't improving that quickly, so getting significant improvement would require more time and resources than we had access to.

D. Iterative Self-Play

For this experiment, we "fixed" a particular Dominion game (i.e. selected the cards in the Supply and kept them the same for each game, and then played the same Learner against itself iteratively. After each game, we trained on the new experience of both the winner and the loser. This had a few advantages. First, the shared model meant we were only improving and updating one thing over time, rather than training tons of Learners only to later discard them. Second, we learned from both winning and losing, which seems valuable. Third, we were able to observe the "loss" in predicting Q-values over time, which seems like a reasonable way to measure whether we are improving. The chart below shows that, after a few games where the loss increases (presumably because all Q-values are initialized to 0, and so "bootstrapping" shows minimal loss because there's little difference), we begin to see it steadily decrease after each game. It seems reasonable to conclude from this that our agent is improving!

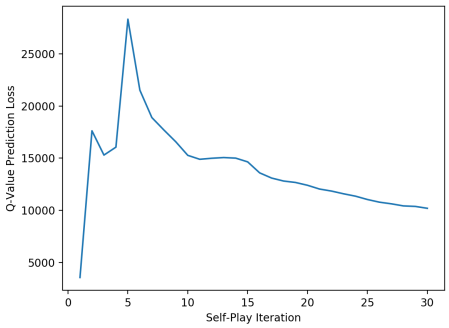

Fig. 3. Average Q-Value Prediction Loss Per Turn Over 30 Self-Play Iterations

V. Discussion

The results of our experiments suggest that a content-associative architecture appears to work well in generalizing the state-action features we extracted. In one early trial (against random agent), one agent stumbled upon the strategy of buying Silvers, then Provinces (the highest-VP card):

```
Game Log:
Player 1 (Computer) bought Silver
Player 2 (Computer) bought Copper
Player 1 (Computer) bought Province
```

The issue with limited training is that games are randomly configured with sets of cards, breeding different strategies. A Learner would have to play many games in order to generalize well, which we unfortunately did not have time to do.

Iterative self training proved to be the most efficient way of bootstrapping training once the Learner could beat Random. We did not train for very long (a day), but the Learner still learns some strategies. We invite you to play with the agent yourself!

Github: https://github.com/garrickf/cs238-dominion

VI. Conclusion

In conclusion, we found that neural networks are good for learning to play Dominion. The network we designed learns generally good things to buy; however, it tends to not be very good with playing actions, probably because of the large amount of experience one would need to understand what the action is useful for. With more time and resources, we would have liked to train more expressive networks with more parameters; and to experiment with extracting more features (such as about the progression of the game, the players overall deck, and information they may know about the other players, such as the VP or treasures they have). These features may be what?s needed for the network to develop more complex strategies.

VII. Attribution

Garrick built the game engine and the deep learning architecture. Ben worked on feature extraction and SARSA implementation, and took the lead on writing this paper.

Acknowledgment

We want to thank Professor Kochenderfer for his enjoyable and well-taught class, *Decision Making Under Uncertainty*. We would also like to thank the community of Synergy House, especially David Gonzalez, for getting the two of us interested in the game of Dominion in 2017.

References

[1] Watkins, C.J.C.H. (1989). Learning from delayed rewards. PhD Thesis, University of Cambridge, England.

[2] Rummery, G. and Niranjan, M. (1994). On-Line Q-Learning Using Connectionist Systems. Cambridge University Engineering Department, University of Cambridge, England.

[3] Mykel J. Kochenderfer (2015). Decision Making Under Uncertainty: Theory and Application. MIT Press, Cambridge, MA.

[4] Vincent Franois-Lavet, Peter Henderson, Riashat Islam, Marc G. Bellemare and Joelle Pineau (2018). "An Introduction to Deep Reinforcement Learning." Foundations and Trends in Machine Learning: Vol. 11, No. 3-4.

[5] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S. & Hassabis, D. (2015). "Human-level control through deep reinforcement learning.
Nature, 518, 529–533.

[6] Silver, D., Huang, A., Maddison, C. et al. (2016). "Mastering the game of Go with deep neural networks and tree search." Nature 529, 484?489

[7] Alqatari, Ammar, Gaiarin, Ben, and Vobejda, Michael. "Agent Q Plays Texas Hold'em."

[8] Brown, Liam, and Crowley, Jeremy. "Perceptron Q-Learning Applied to Super Smash Bros Melee."