Reinforcement Learning Report: Catan Bot

Ryan McGuinness

5/10/2022

Introduction and Motivation

Reinforcement learning algorithms have been steadily increasing in popularity recently. The basic principle of all RL algorithms is really quite simple: force the agent to take an action, and if the action was beneficial then give the agent a reward – it is then the agent's goal to maximize the reward it receives over time. If this at all reminds you of the way one might train their dog to behave around the house, you are not alone. But one important distinction to note between these agents and house pets is that even though the agents are given the reward immediately after their action, the RL agents do not greedily maximize the reward they receive, but instead look to maximize rewards over the whole training cycle, meaning they will take actions that forgo immediate rewards if it comes with better prospects of being rewarded in the future.

One convincing reason to use reinforcement learning techniques is that training data is not required to build an RL agent. Instead, the agent is put in an *environment* where it will make actions and evaluate how beneficial the action was. Actions that are deemed beneficial will be more likely to be taken again while training, and when evaluating the agent will select the action it believes to be most beneficial.

Catan is one of the world's most popular modern board games, where players act as settlers in a new land in competition to grow their territory quicker than the others. Although the inclusion of dice and cards make many believe Catan to be a game of chance, there is much more to the strategy of the game. Due to its popularity and its strategic depth, I choose this game as the environment to implement the Deep Q-Learning Network (DQN) and Proximal Policy Optimizer (PPO) reinforcement learner agents.

Theory

First, we must start with some basics of reinforcement learning. As alluded to in the the introduction, RL agents work by taking actions and calculating the *reward* for the resulting state. The action the agent takes is determined by the state of the observed environment and the agent's *policy*, which is responsible for mapping that observed state to an action. The reward is (or is not) given immediately after taking the action and observing the resulting state, nd information about the previous state, the action, and reward are used to update the policy. But it is not always right to look so short-sighted at only the next state. Therefore, agents employ a *value function* responsible for maximizing all rewards given over time.

Before we can understand Deep Q-Learning, we must start with standard q-learning. At the start of the q-learner's training, the agent initializes the *Q-table*, which has dimensions of the number of possible states and number of possible actions, to all zeros.

State	Action 0	Action 1	Action 2	...	Action N
State 0	0	0	0	...	0
State 1	0	0	0	...	0
State 2	0	0	0	...	0

State	Action 0	Action 1	Action 2	...	Action N
...	...	...	...	...	...
State M	0	0	0	...	0

The values of these tables are *Q-values*, representing the expected long-term reward accumulation for taking an action in a specified state. The more favorable an action is in a specific state, the higher the corresponding q-value. Then, when the model must select an action, it simply chooses the action with the maximum q-value for the state. The agent observes the new state and updates the q-value according to the following relation:
$Q(s_t,\ a_t) \leftarrow Q(s_t,\ a_t)(1-\alpha) + \alpha\ \gamma\ r_t\ \max_a\ Q(s_{t+1},\ a)$
where s_t represents the current state, s_{t+1} the resulting state, a_t the action taken by the agent, r_t the reward given to the agent, α the learning rate, and γ the discount factor ($0 \leq \gamma \leq 1$ to give current rewards more importance than possible future rewards).

However, this simple behavior could lead to some issues. It is true when you are evaluating the model, that you would want the agent to choose the "best" (q-maximum) action. However, when training the agent, it is important to explore as much of the action space as possible. The point of this is that if the first action we choose by chance when the q-table is initialized to zeros happens to come with a reward, then the agent will always choose that action in that state if there was nothing to stop it. Even though the action may be suboptimal, the agent would never know. Therefore, we introduce a parameter ϵ, the chance for the agent to choose a random action, regardless of state. It is often advisable to let ϵ decay over the training time, as after most of the action space is explored, it becomes more important to explore the most promising actions more in depth.

The underlying assumption of q-learning is that the best action to take depends only on the current state of the environment, and no past states are relevant. This is the Markov property, making q-learning Markov Decision Process. However, the disadvantage of q-learning is that when the problem space is highly complex, as is the case for Catan, computing the q-value for all possible combinations of state and action is very expensive, but if not done completely then the agent will likely fail to make the correct decision in an unfamiliar state.

It is this problem that Deep Q-Learning aims to solve. Building a Deep Q-Network (DQN) model involves training a neural network to map environment states to the q-values for its actions. When faced with an unfamiliar state, q-learning cannot infer optimal actions of similar states, but we can approximate the q-values with a neural network.

In the DQN, the loss function is the mean square errors of predicted Q-value and Q^*, the target q-value: $Q^* = r_{t+1} + \gamma\ \max_a Q(s_{t+1},\ a)$. In this manner, DQN becomes a regression problem.

I also utilized Proximal Policy Optimizer (PPO) model, which uses a policy gradient optimizer, commonly implemented in the Actor-Critic Model, which uses two neural networks, one which handles the actions (the actor), and one which handles the rewards (the critic). While this was probably worth studying more in depth, it is a relatively new advancement in reinforcement learning, and I found the literature much more difficult to comprehend than DQNs. As the goal of this project is to expand my understanding of deep learning (and machine learning in general) to encompass reinforcement learning techniques, I am quite happy leaving the details for this specific implementation at this.

Methods and Implementation

Catan, like most board games, is competitive in nature. In Catan, the first player to reach 10 Victory Points (VPs) wins, so our models must try to accomplish just that. But in order to build an RL agent to play Catan, we need other agents to pit the learner against. But how can we train our *first* agent to play Catan against an RL model? One solution would be to build an random agent, that makes a random legal play every time we ask it to make an action. While this solution is entirely reasonable, I went with another approach: why not build a model to play Catan cooperatively? Instead of trying to win, we can make our

first agent's goal to be maximize the total number of Victory Points (VPs) on the board, so that it doesn't have to play against any other models. Then, we build our first competitive agents to play against this one.

This is not nearly as unreasonable as it sounds. Sure, the cooperative agent may try to make actions that favor its opponents when they are behind, but if the competitive model has a lead, then the cooperative model will be forced to build up its own VPs to maximize the total VPs on the board. Still, we shouldn't expect the cooperative player to be much better than a random player at actually winning the game. But once our model gets good at defeating the cooperative model, we train a second competitive agent to play against the first. This process can be iterated indefinitely, in theory creating better and better players each time. In this manner, I created thirteen different reinforcement learning agents before finally creating a DQN agent to play against the RL model.

The first several competitive models were all PPO learners. We shouldn't expect too much from the PPOs considering the scale of the observation space, but we should expect one to be better than random, and in total 13 were trained consecutively, each one beating the last in performance. The final one of these PPO models were then used to as the opponents of the DQN model.

The `gym` package from OpenAI is used to create the environment. Conceptually, this environment represents the game of Catan, and is responsible for enforcing the rules and carrying out the actions the agent elects to take. The environment has an "observation space", the space of all possible states a game of Catan can take (or, at least all the states we care to inform our model about), and an "action space", the space of all possible actions the agent can take. The observation space used in this environment carries information about all buildings on the board and all cards in player's hands (while having information about cards in other hands is cheating in most games, in Catan a player can see most cards entering his opponents' hands, and an experienced player can deduce with great accuracy the otherwise unknown cards). The environment also is responsible for calculating rewards for the agent. In my environment, playing an action which results in gaining VPs gets a small reward, but winning the game comes with a much larger reward that overshadows the simple VP reward. However, this reward for winning is reduced every turn that is played. The idea is that an agent just starting to train will be incentivized to gain VPs, but once it wins, even if by accident, it will look for ways to win, and win quickly.

The `stable-baselines3` package (based on OpenAI's `baselines` package) is used to create the agent. This agent is responsible for making actions and updating the policy. For the first "cooperative" model and the first batch of "competitive" models, a PPO model was used. A full-fledged DQN model was built to play against the best PPO model, and finally a hybrid model that uses PPO to take most actions but DQN to decide on settlement locations was employed.

Some simplifications to the rules of Catan were used in the environment. Limits on the number of cards in play and number of settlements on the board were not enforced, and while trading with the bank was allowed, trades between players were off limits. Most significantly, the starting setup of the board was predetermined - there are ~3.7 quadrillion ways a Catan board can be set up before any player takes his first turn, and I'm simply not prepared to think about such number of possible states without any actions taken, so I chose to use the beginner's board setup that the Catan rules recommends. Credits to the `PyCatan2` package by josefwaller for doing much of the heavy lifting with the Catan game logic.

Results and Conclusions

The 100% DQN model did not perform well in training, and in fact had impressively bad results, coming in last in nearly every single game against the PPO players. It isn't totally unexpected that PPO produced the better player. It seems that even with deep learning, the number of possible states a game of Catan can be in is simply too large for the Q-learning network to handle, even with a neural network layered on top of it. However, I next tried limiting the DQN usage to selecting settlement locations and used PPO to perform the rest of the actions. While this severely limits DQN usage, as there are relatively few choices to really be made here, I did see this "hybrid" model win 4-player games over 30% of its games against the PPO models alone, and came in last just 19% of games against the nearly identical PPO models.

I would still like to expand the DQN agent's scope to include choosing when to build versus choosing when to pass the turn. The fact that the pure DQN did not perform well is discouraging, but this project was fun enough that I can imagine working out more of the kinks in my downtime. It is unfortunate that I had such sparse luck with the DQN model before this deadline. However, that the hybrid model wins more often than its opponents is an encouraging sign for the use of DQN in the Catan Bot. Aside from experimenting with more DQN usage, the next steps for this project include experimenting with more board setups aside from the beginner's board, and making a UI to easily pit this bot against humans.