Investigating Optimal Choice of Adversary in Deep Q Learning Using Nim With Cash

Joshua Twitty supervised by William Gasarch [1]

Abstract

The use of computers for two-player zero-sum games has been a subject of interest since the concept of Artificial Intelligence was first posited, with early focus being on programs able to play Chess. Today, many strategies exist for creating programs for such games. One such method is framing the problem of playing a game as a reinforcement learning problem and then employing techniques from that field to design a game playing agent. In doing so, the problem of where to source the training data arises. In the case where the game in question has a computable optimal strategy, one could create training data by having the agent train against an opponent employing this optimal strategy, though this introduces an exploitation vs. exploration problem between the agent learning how to beat a player employing the optimal strategy and being able to beat players using diverse strategies. This paper explores the effects of varying training adversaries in the novel domain of Nim with Cash on a neural network player trained using Deep Q Learning.

1. Introduction

1.1. Nim With Cash

Nim With Cash is a variant of the combinatorial game Nim that was proposed by Gasarch in (Gasarch et al., 2015)

In regular Nim, there is a pile of n stones in the board. Each player alternates turns by removing a number of stones (traditionally 1, 2, or 3) stones from the pile. The first player that is unable to remove a stone due to the number of stones on the pile being less than the smallest number of stones they are allowed to move loses. (Another variant of the game has the player who is forced to take the last stone losing, but this version of the paper does not discuss this version.)

If the number of stones the players are allowed to move from the tile are 1, 2, or 3, there is a known dominant strategy for this version of Nim for a player who has a turn where the pile has a number of stones that is $0 \not\equiv \mod 4$. If the number of stones in the pile is congruent to $m \in \{1, 2, 3\}$, then the strategy is to remove m stones. As a consequence, in a Nim game with optimal players, player 1 wins a game of Nim if and only if the number of stones in the pile at the start is not congruent to 0 mod 4.

Nim with Cash is a variant of Nim where each player is given a finite integer (cash) of stones that they are allowed to remove over the course of the game. An additional loss condition is added wherein if a player cannot move due to lack of cash they will lose. The amount of cash that each player has need not be symmetric. With enough money for both players, this game devolves into Nim, and with very little money the best strategy is to simply remove as few stones as possible per turn so as to save cash. In the in-between cases, the strategy becomes more varied. The function $W(N, a, b)$ representing the winner of a game of Nim with Cash where player 1 has a cash and player 2 has b cash can be computed via the following relationship:

$$\begin{cases} 1 & W(N - a_1, b, a - a_1) = 2, \\ & W(N - a_2, b, a - a_2) = 2, \\ & W(N - a_3, b, a - a_3) = 2 \\ 2 & O.W. \end{cases}$$

Where a_1, a_2, and a_3 are integers representing the amount of stones a player is allowed to take from the pile in a turn. Using this observation, one can determine the winner of a game given the current state and also identify what move the winner needs to take to achieve victory.

1.2. Reinforcement Learning and Deep Q-Learning

Reinforcement Learning is a setting in machine learning wherein one attempts to teach an autonomous agent to perform some task in an environment by utilizing a reward system that biases it towards rewarding actions and shifts it away from actions that are unrewarding or penalizing. Reinforcement learning is done over an environment referred to as a Markov Decision Process (MDP), consisting of a set of states S, a set of actions in each state A_s, and a transition function $P_a : S \times A_s \to S$ that determines the resulting state after an action is taken (and can be stochastic), and

a reward function $R_a : S \times A_s \to \mathbb{R}$ that determines the reward that is given for transitioning from a state s to state s' given action a.

The function that an agency uses to select an action given a state is called the policy and is traditionally denoted by $\pi : S \to A$ (this function need not be deterministic). Reinforcement Learning algorithms typically seek to find a policy that maximizes the total reward across some finite time horizon, called an episode. For a two-player zero-sum game like Nim or Nim with Cash, the natural episode is just a single game. The reward function that we will use will dispense a reward for each move the player takes- which will be a reward of 1 if an action results in winning the game, a -1 if it loses the game, and 0 if the game is not over.

Q-learning is a basic Reinforcement Learning algorithm that functions by approximating the state-action value function $Q : S \times A_s \to \mathbb{R}$. The Q function is defined as

$$Q(s, a) = R(s, a) + \gamma \max_{a \in A_{s'}} Q(s', a)$$

where s' is the state that is landed in after executing action a at state s (if this could result in different states due to the environment being stochastic, then this becomes an expectation over all of those states) and $\gamma \in [0, 1]$ is known as a discounting factor that constrains the function to focusing on more immediate rewards. Intuitively, the recursively-defined $Q(s, a)$ can be thought of as the expected reward if one chooses action a in state s plus the (discounted) expected reward if one picks the action that results in the highest future rewards from the state that action lands the agent in. With access to the Q function, one could maximize the reward over an episode by following a policy that greedily selecting the action that maximizes the Q value at the given state.

In some environments, the actual value of Q for each state-action pair can be computed using dynamic programming. In larger environments, it becomes more practical to compute an approximation of the Q function. The best known algorithm for doing so is named Q-learning. Q-learning approximates the function Q by storing an approximation of it as a lookup table using a representation of the current state and the action in question as the key and the current approximation of Q for that state-action pair as a value. Initially, this table might be initialized with all zeroes, but by simulating an episode in the environment and observing rewards, the values in this table can be updated via the equation

$$Q^*_{new}(s, a) = Q^*_{old}(s, a) + \alpha(r + \gamma \max_{a \in A_s} Q^*(s', a))$$

where Q^* is the table's current estimation of the Q value and $\alpha \in [0, 1]$ is the learning rate, a hyperparameter that controls how much the agent's evaluation of action choice in a given state should change based off the observations from simulating one episode. Intuitively, a high learning rate makes the network learn faster but be more unstable due to changing its value evaluations heavily based off little data, while a lower learning rate makes the agent learn a policy more slowly, requiring more training, but potentially landing on a better policy due to improved stability.

When training an agent using Q-learning, the agent learns through exploring the environment. As such, the agent requires a policy to use as it explores during training. It is desirable to have an agent that learns every iteration of training and iterates on past successes, but also one that tries out new strategies that may prove to be more effective than the ones it has already found success with. This introduces a conflict between exploring the environment and finding new strategies and exploiting strategies that are already known to be good. This is known as the Exploration/Exploitation trade-off. In order to handle this trade-off, the policy that is typically used during training for Q-learning is the ϵ-greedy policy. Parameterized by $\epsilon \in [0, 1]$, this policy allows the balances exploration and exploitation during training. Denoting the ϵ-greedy policy as π_ϵ, it is defined as follows:

$$\pi_\epsilon(s) = \begin{cases} \max_{a \in A_s} Q^*(s, a) & Pr(1 - \epsilon) \\ \text{Uniformly at random select action} & Pr(\epsilon) \end{cases}$$

Where Q^* is the current approximation of Q at any particular step of training. ϵ can be understood as a hyperparameter that controls how often the agent explores during training versus choosing actions it knows to be good, with a high value of ϵ causing the agent to select more actions randomly and thus explore more, and a lower value biasing the agent towards actions it has already found to be good.

Deep Q-learning simply replaces the lookup table in Q-learning (which can get very big) with a neural network that can approximate Q instead. These neural networks can take in a representation of the state in the form of an array or tensor and then output an array the size of the number of actions available, with the $i$ith entry in the output array representing the approximation of the Q value for the ith action. The network is trained using backpropagation using the disparity between observed Q value of a state and the network's prediction of those rewards from each state as the loss. While in this case we could compute the true value of Q using a lookup table, in this experiment we are interested in the algorithm's ability to compute the Q through playing the game and the effects on changing the opponent it trains against. To this end, the target value of $Q(s, a)$ is

$$r + \max_{a \in A_{s'}} Q^*(s', a)$$

where Q^* is the network's current prediction of Q and r is the reward the agent received from the environment upon

picking action a in state s. Empirically it has been observed that using the network's own prediction of Q for the loss creates instability in training due to the function it is meant to learn changing every iteration, so in this paper a separate network called the target network is used for $Q*$. The network representing the agent's policy is then referred to as the policy network. The target network has the same parameters as the policy network did some number of steps ago, making the function the network is meant to predict slightly more stationary. After training is over, the agent makes decisions by taking the state as an input to the neural network, then choosing the action with the highest predicted Q-value in the output array.

1.3. Nim with Cash as an MDP

A two-player zero-sum game can be cast as an MDP by creating an opponent and having the agent play the role of the player playing against that opponent. The state space is the state of the game, the action space is the set of legal moves in a space, the transition function can be simulated by having the agent take the action it selected, having the opponent take its turn afterwards, and then returning the resulting state. As discussed before, the reward function can simply return a 1 for moves resulting in winning states and a -1 for moves resulting in losing states.

One choice that can be made during the training process for Q-learning in such a game is how to source the training data. As stated in the earlier section on the game, it is possible to create a player that chooses the dominant strategy whenever one is available using dynamic programming. One possible way of producing training data for the agent is simply having it play against the perfect player to generate the data. However, depending on the goals of the agent, this can potentially be another source of an exploitation vs. exploration trade-off. If one's goal is to create a player that performs generally performs well against a variety of players, then there are game states that it may not see and create a value estimation for if it trains exclusively against a deterministic player that picks the most optimal option at each game state.

One possible solution for this problem is to borrow the epsilon-greedy approach used for the agent's training policy. That is, the opponent will take their regular action with probability $1-\epsilon$ and a random one with probability ϵ. Doing this could expose the agent to more game states and makes the environment it is learning stochastic, making the agent more able to cope with opponents employing various strategies. However, this also introduces yet another hyperparameter- the choice of ϵ for the adversary. The primary goal of this paper is to investigate the effects of the choice of ϵ when this strategy is employed in the Nim with Cash domain.

2. Methodology

This experiment uses a two-layer, feedforward neural network implemented using the Pytorch library. The input state is a one-dimensional tensor of size 3 representing the game state containing the number of stones in the pile, the agent's cash, and opponent's cash respectively. The agent is then trained using the Q-learning strategy described earlier in the paper. The parameters used during training were:

$$\alpha = 0.1$$
$$\gamma = 0.9$$
$$\text{\# of training epochs} = 5000$$

For the value of ϵ, a decaying value of $\epsilon(t) = 0.3+(0.7)e^{\frac{t}{200}}$ was chosen in order to have the agent start off prioritizing exploration at the beginning of training and gradually switches over to exploitation of known strategy as training proceeds.

The network was then trained against players equipped with a lookup table for the optimal action that follow a variant of the ϵ-greedy policy similar to the one described earlier:

$$\begin{cases} \text{Take optimal action} & Pr = 1-\epsilon \\ \text{Take random action} & Pr = \epsilon \end{cases}$$

The value of ϵ for these players was varied by increments of .20, with one agent trained per value. Trained agents were then evaluated by how well they performed against players following the strategy above for $\epsilon \in \{0.0, 0.2, 0.4, 0.6, 0.8, 1.0\}$.

Because the results of training were affected heavily by randomness, several trials were conducted for each game the network learned to play.

As a test to whether the observed results generalized to other variants of Nim, the network was also trained in the same fashion in the player 1 position of Nim with 101 stones. After a few lackluster trials with the same architecture as the Nim with Cash network, however, the architecture for the network was thinned down to 30 neurons in the hidden layer instead of 100.

3. Results

The results of the experiments can be viewed in the Appendix.

4. Discussion

In general, this environment appears to be an unstable one for deep Q-learning and it is possibly for a training session to result in a policy that performs worse than random, or not much better than random (i.e. performance against a

random player 50%). When this did not happen, the policy the player converges to seems to defend on the value of epsilon.

When the policy converged to a better-than-random value when trained on an adversary with lower values of epsilon, the agent was unable to perform very well against the agent it trained against but could perform very well against opponents with higher values of epsilon. This makes sense intuitively, as higher values of epsilon result in a less optimal opponent and should thus be easier. When choosing a higher value of epsilon the effect where the agent would perform better on downstream opponents is less pronounced, but one can observe the agent performing worse against players with lower values of epsilon. In general, it seems like a few wins against adversaries with lower ϵ values leads to better performance than winning against adversaries with higher values of ϵ.

No agent was able to beat the perfect player in either instance of Nim with Cash. For the (100,55,55) game this is because player 2 has a dominant strategy in this instance. For the (100,60,55) instance, I presume this is just because the only policy which can do this is the optimal policy and this was difficult for the algorithm to converge to.

As can be seen in Table 7, the algorithm seems to perform worse on regular Nim than Nim with Cash- this is possibly due to the state space being much simpler (one-pile Nim's state is just the number of stones on the pile) and thus having less information for a neural network to work with. The training never seemed to converge to anything better than random.

5. Conclusion

In the Nim with Cash domain, the quality of an agent trained using this strategy appears to be deeply sensitive to the choice of ϵ for the adversary. In particular, adversaries with lower values of epsilon produced agents that could defeat values of higher epsilon (provided that the training did not fail to converge). In the case of using Q-learning in this domain, it appears that higher difficulty adversaries make better training partners in general.

The Nim with Cash domain is difficult for an agent to learn the optimal policy of using Q-learning, but it appears from the results of these experiments that one can get a Q-learning agent to perform better than random. The biggest issue Q-learning faces in this domain is failing to converge, which it seemed to do fairly often.

There are several future research directions that can continue off this paper- include using more sophisticated reinforcement learning algorithms, such as Actor-Critic methods. Another direction would be performing this experiment and looking at long-term data for how often training fails to converge (i.e. maintains about or less than 50% win rate against the fully random player) against each adversary. This experiment didn't collect such data, but it would expand conclusions about which adversary is best to train on if there were some adversaries training was statistically more likely to converge against. One could also check to see if one can utilize techniques not used here, such as Experience Replay (which is often talked about in Deep Q Learning related literature and tutorials) to improve the likelihood of training converging. Finally, one could also look at other variants of Nim such as multi-pile Nim, multi-pile Nim with Cash, or variants of Nim with different action spaces (e.g. single-pile with removing 1, 3, or 5 stones instead of 1,2, or 3 or multi-pile where each pile has a different number of stones you can remove). One could check to see if the patterns observed here carry over to these other games.

References

Gasarch, W., Purtilo, J., and Ulrich, D. Nim with cash, 2015. URL https://arxiv.org/abs/1511.04035.

6. Appendix

ϵ	0.0	0.2	0.4	0.6	0.8	1.0
0.0	0.0	41.5	92.0	99.9	100.0	100.0
0.2	0.0	0.3	5.5	21.3	38.0	51.2
0.4	0.0	0.0	0.0	0.0	0.0	0.9
0.6	0.0	0.0	0.0	0.0	0.0	0.7
0.8	0.0	0.0	0.0	0.0	0.0	0.4
1.0	0.0	36.2	94.5	99.9	100.0	100.0

Table 1. (100,55,55) Trial 1. The y-axis is the value of ϵ for the adversary the agent trained against, while the x-axis is the value of ϵ for the adversary fought against post-training. The numbers in the cells represent the win% of the agent after 1000 games against that opponent.

ϵ	0.0	0.2	0.4	0.6	0.8	1.0
0.0	0.0	36.1	93.4	100.0	100.0	100.0
0.2	0.0	33.7	95.0	100.0	100.0	100.0
0.4	0.0	0.0	1.0	13.4	35.7	53.9
0.6	0.0	0.0	1.3	14.4	37.8	54.3
0.8	0.0	23.5	45.2	82.8	96.3	99.9
1.0	0.0	0.1	6.5	21.6	35.8	49.7

Table 2. (100,55,55) Trial 2

ϵ	0.0	0.2	0.4	0.6	0.8	1.0
0.0	0.0	36.9	94.6	100.0	100.0	100.0
0.2	0.0	10.1	20.8	29.5	44.4	55.3
0.4	0.0	34.4	87.9	99.9	100.0	100.0
0.6	0.0	0.4	9.1	23.4	31.8	41.4
0.8	0.1	34.2	93.5	100.0	100.0	100.0
1.0	0.0	35.6	94.9	99.9	100.0	100.0

Table 3. (100,55,55) Trial 3

ϵ	0.0	0.2	0.4	0.6	0.8	1.0
0.0	0.0	0.2	3.1	16.2	31.0	38.8
0.2	0.0	1.7	10.1	25.8	30.9	37.6
0.4	0.0	39.7	92.5	99.8	100.0	100.0
0.6	0.1	0.6	14.6	31.7	40.2	53.4
0.8	0.1	12.2	70.3	98.5	100.0	100.0
1.0	0.1	26.4	89.1	99.6	100.0	100.0

Table 4. (100,60,55) Trial 1

ϵ	0.0	0.2	0.4	0.6	0.8	1.0
0.0	0.0	1.9	10.4	21.7	32.2	49.3
0.2	0.0	0.0	0.1	0.3	4.5	16.6
0.4	0.0	1.8	17.1	31.4	44.2	53.2
0.6	0.1	4.8	16.0	25.8	37.7	66.9
0.8	0.0	1.3	15.9	31.8	39.7	52.8
1.0	0.0	35.6	93.8	99.9	100.0	100.0

Table 5. (100,60,55) Trial 2.

ϵ	0.0	0.2	0.4	0.6	0.8	1.0
0.0	0.0	6.8	14.5	28.9	40.8	53.2
0.2	0.0	1.7	15.9	30.4	40.7	53.2
0.4	0.0	0.0	0.0	0.6	3.9	17.9
0.6	0.0	2.0	16.3	28.4	38.8	57.7
0.8	0.0	3.0	16.1	25.0	39.1	53.7
1.0	0.0	30.7	36.4	63.3	91.1	99.7

Table 6. (100,60,55) Trial 3.

ϵ	0.0	0.2	0.4	0.6	0.8	1.0
0.0	0.0	16.3	31.9	43.8	53.9	66.7
0.2	0.0	11.0	19.5	26.4	35.8	43.5
0.4	0.0	9.1	23.6	33.3	44.2	52.2
0.6	0.0	7.9	17.7	27.3	31.0	35.9
0.8	0.0	10.7	18.2	27.3	30.6	37.2
1.0	0.0	9.5	17.7	27.5	33.6	39.0

Table 7. Regular Nim(101)