Creating Ramsey Graphs Using Neural Networks

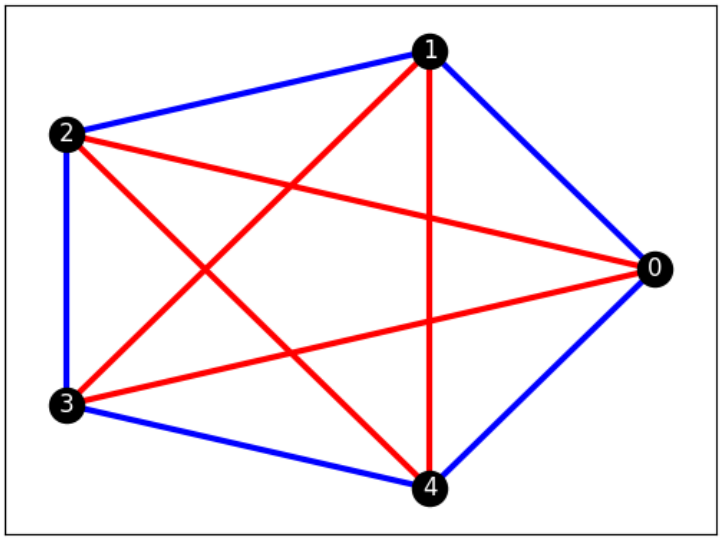
Jyolsna Sunny

July 2023

1 Introduction

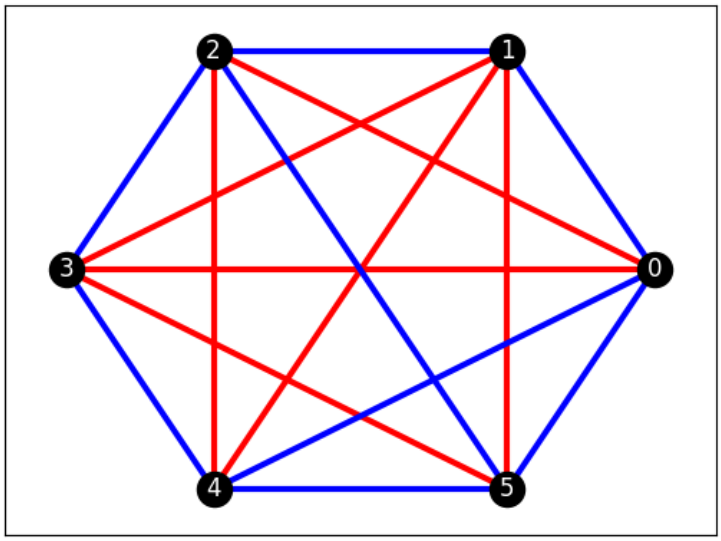
1.1 Ramsey Theory

In graph theory, Ramsey Theory encompasses a wide variety of ideas, but we are interested in the Ramsey number R(k,k). R(k,k) denotes the minimum number of vertices, N, needed to guarantee that, in a two-color edge-coloring, the complete graph of size N contains a monochromatic clique of size k. Consider R(3, 3). It is possible to two-color edge-color a graph on 5 vertices such that there is no monochromatic triangle, as seen below.



However, it is not possible to do the same on a graph with 6 vertices. A graph with 6 vertices must contain a monochromatic triangle, which means that

R(3, 3) = 6. An example can be seen below, however it is worth noting that an example is not sufficient to prove that every graph on 6 vertices contains a monochromatic triangle.



1.2 The Game

Let's play a game. There are two players, player red (1) and player blue (2). Our board is an uncolored complete graph on N vertices. Starting with player red, each player takes a turn coloring one edge of the graph in their respective colors with the goal of forming a clique of size k in their color. Suppose we could train the players to become really good at the game and pitted them against each other. They would supposedly learn to block each other's cliques as they try to form their own. In order to show that R(k,k) > N, we need to color a graph on N vertices with no monochromatic clique of size k. This corresponds nicely to a the game resulting in a draw. If the players learn to play until the board is full with neither player forming a clique, then we have found a graph with no monochromatic clique.

But what benefit does posing this problem as a game give? Theoretically, we could iterate over all possible colorings of the graph on N vertices, but that would be a discrete search which would be very computationally expensive. In the worst case scenario, we would have to look at every single coloring before finding one that one that has no monochromatic clique. Therefore, instead of iterating over graphs, another approach would be iterating over players. The

sample space of players is smooth in terms of its parameters, allowing us to use gradient descent to try to find the ideal player. Much like search an sorted list vs. and unsorted list, searching a continuous sample space is much more efficient than searching a discrete sample space. Therefore, posing this problem as a game provides a promising opportunity to more efficiently find a significant graph coloring.

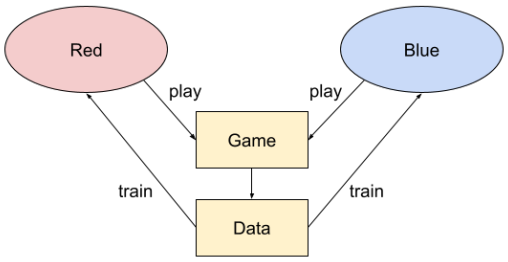
1.3 Reinforcement Learning

We want to train a neural network to play the game, with the goal that they get good enough to play each other to a draw. As with any machine learning problem, we first need to define the input and output space. The input space are the features we want to take into consideration, namely the current state of the board and the possible actions the current player can take. The output space is what we want to determine, namely the utility or value of a given move. The problem is that we do not know what the utility of a given state and move should be. We could easily train a model to predict values if we had a data set of states and actions and their respective values, but we do not. So what do we do? We create that data set!

Reinforcement learning is a machine learning training method that rewards wanted behaviors and punishes unwanted behaviors. In this case, we want to reward a network for making a good move (such as forming a clique) and punish the network for making a bad move (such as not preventing the opponent from forming a clique). We can reward or punish the network by giving the player points or taking away points. But how do we determine what is a good move and what is a bad move, especially among the moves in the middle of the game? And thus begins our journey into figuring out how to do that.

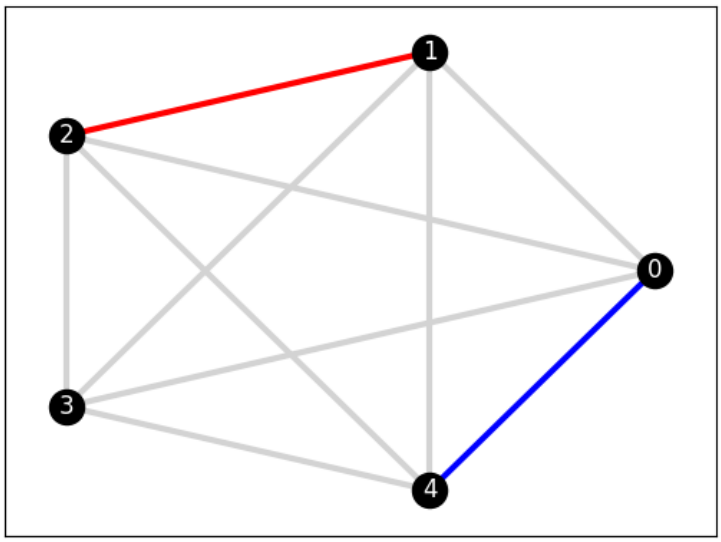
2 Experiments and Results

2.1 Standard Q-Learning Network



Consider a data set containing {current_state, action_taken, next_state, immediate_reward, terminal_flag} for each row, shortened to {s, a, s', r, f}. The states are represented by an adjacency matrix with a value of -1 representing a red edge and +1 representing a blue edge. The action is a one-hot adjacency matrix, with a +1/-1 for the action to be taken, and 0 for the remaining entries. The reward is statically set as 1 point for any move, 100 points for the winning move, and 0 points if the game ends in a draw.

For example, consider the following game state.



The adjacency matrix of the current state could be represented as

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & -1 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Suppose the next red move was the edge between vertices 2 and 3. We could represent that move as

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 \\ 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

We pass these two parameters to our network, which outputs a single scalar value for the utility of that given move played on that given game state.

We begin with two networks, one for the red player and one for the blue player. What we want our network to predict is, not just the immediate reward, but the reward we may get in the future by playing a certain action. As such, we come to our loss function.

$$Loss = (Q(s_t, a_t) - (r_t + Q(s_{t+1}, a_{t+1}))^2$$

In other words, we want to minimize the difference between our network's predicted utility and the sum of the immediate reward and our future rewards.

With this model, we hope that reward values will propagate backwards from the winning move, following the track of moves that led there. Roughly, the plan is

```
Intialize Q–Network
Simulate games to generate data
Training Loop:
    Grab data batch
    Estimate utility with Q
    Given (s, a)
        Current: Q(s, a)
        Improved: y = r + maxQ(s', a')
    Minimize (Q(s,a)−y)^2
Simulate more games and retrain
```

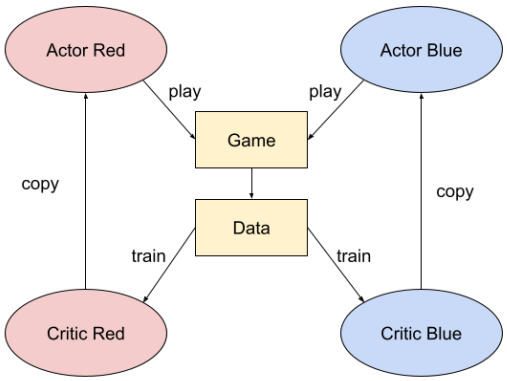
There are a few changes that can be made to theoretically improve the performance of this algorithm. One such change is implementing an epsilon-greedy algorithm. In order to address the exploration vs exploitation problem, we can decide to simulate games such that each move is either chosen randomly or using the network with some probability epsilon. Initially, we want a high degree of random moves being played (exploration), so we set the probability of random moves higher. Slowly, we decrease that probability and move towards an almost-pure network game (exploitation). The benefit of this strategy is to create a balance of learning new moves and perfecting the ones you know. Also, this avoids the network from learning to play the same game over and over, which wouldn't give us much to learn from. We can also extend the idea behind the epsilon-greedy algorithm to the evaluation of our updated Q value like so

$$updatedQ = 0.9 * currQ + 0.1 * nextQ$$

This allows us to modify how much we value the reward we know vs the reward we are predicting. And similar to the game simulation, this balances exploration and exploitation in the evaluation of Q values.

I have yet to discuss network architecture, largely because, with a complex enough network, it will have minimal impact of the predicted values. But one potentially positive change made to the network architecture is to add a convolutional layer that filters out each row/column of the adjacency matrix, effectively isolating the information for a single vertex.

2.2 Actor-Critic Network



One of the main issues with a standard Q-learning network is that the value we want the network to return is always changing. It's harder to train a network on a moving target. So what if held the target still? Consider networks A and B. The idea is to use one network as the target for the other. So, in this case, with every training pass, we would use one network to evaluate for a move but train the other network.

Consider 4 networks : actor_red, critic_red, actor_blue, and critic_blue. The actors plays the role of evaluating which would be the best move to take. The critics are trained using those evaluations, almost like someone studying players playing a game.

$$updatedQ = r + maxActor(s', a')$$
$$minimize[Critic(s, a) - updatedQ]$$

The effect of this is that it creates a still target for each of the critics to learn off of since the actors' actions are consistent. The critics are learning the value of actions chosen by the actor, and once those values are learned, we use that as the basis for choosing new actions. After some training in this method, we copy the critics to the actors so the critics can play using what they have learned, and we start again.

One way to possibly improve performance with the actor-critic network is to introduce staggered training. Instead of copy both critics to both actors, copy

them one at a time. The idea is that each player will learn to play against an adversary who has a static strategy, instead of trying to learn against a player whose strategy is always changing.

When looking at the loss on a given batch, we see that the loss greatly varies. Losses on moves near the beginning or end of the end have very small loss while the moves in the middle vary. One way to combat this inequality in training is to use priority batching. When getting a batch of data, we can choose the batch probabilistically based on which rows have the largest current loss, and train our network farther using those.

3 Analysis

3.1 What We Did

In implementing these networks, we used a simple network architecture and trained it to play of a complete graph of 8 vertices (so that the data set is smaller and the network can train faster). The actor-critic network proved to be the most promising. It was successful in learning how to play to a draw approximately 40% of the time on a small graph. However, it proved less promising on a larger graph with more vertices. Perhaps it was simply so slow to train that we did not see any meaningful results or perhaps this strategy does not effectively extend to larger examples.

3.2 What We Can Do

A simple, possibly promising change to make would be create a more complex network architecture. It is entirely possible that the network architecture used was just not complex enough to pick up on essential features. As per Chaos Theory, perhaps the features we need to learn are so infinitesimally sensitive to small changes that we could not detect them with a simple neural network.