Learning to Play the Worker-Placement Game

Euphoria using Neural Fitted Q Iteration

Matthew Yeaton*

May 20, 2014

Abstract

We design and implement an agent for the popular worker placement and resource management game Euphoria using Neural Fitted Q Iteration (NFQ), a reinforcement learning algorithm that uses an artificial neural network for the action-value function which is updated off-line considering a sequence of training experiences rather than online as in typical Q-learning. We find that the agent is able to improve its performance against a random agent after only a relatively small number of games.

I. Introduction

Since the nascent days of machine learning and artificial intelligence, board games have served as a useful and fun testing ground for various concepts and techniques in these fields. This is unsurprising when we consider the perks of board games for this purpose, namely that board games may be made arbitrarily complex while still maintaining well-defined inputs, beginnings, and ends, and with clear measures of success and failure. As such, board games can provide an ideal testing ground for noisier “real world” tasks in a safer, cleaner setting.

Furthermore, board games are fun! Board games have seen an astounding resurgence in the past decade or two, so thinking about better ways to design competitive AI agents for direct consumers to play against may be a valid goal in its own right.

Here, we discuss the design and implementation of an agent for the popular board game Euphoria using Neural Fitted Q Iteration.

A. Reinforcement Learning and MDPs

Reinforcement learning can best be understood in contrast to supervised learning. Whereas in supervised learning the learner receives a labeled dataset, in reinforcement learning the learner (or agent) must generate its own data by interacting with its environment and receiving the associated penalties and rewards. In the case of board games, interacting with the environment entails repeatedly making moves, observing the outcome, observing the

*Matthew Yeaton. Email:mry237@nyu.edu.
outcomes of your opponent’s moves, and so on until the game has ended, generally only receiving the reward at the end of the game.

Reinforcement learning problems, and by extension board games, are typically modeled as Markov Decision Processes (MDPs). A MDP consists of a set of states $S$, a set of actions $A$, where $A_s \in A$ is the set of actions available in the state $s$, transition probabilities

$$\Pr[s'|s, a], s, s' \in S, a \in A$$

and reward probabilities

$$\Pr[r'|s, a], r' \in \mathbb{R}, s, s' \in S, a \in A$$

The objective of the MDP is to find an optimal policy

$$\pi^*: S \rightarrow A$$

that maximizes the cumulative rewards for each state. Because of the Markov property, we need only find an optimal policy for a given state, and thus do not need to consider the entire history of state-action transitions but only the best action for the given state. Board games can be modeled as finite horizon MDPs with finite (but potentially large) state sets, $S$, and action sets, $A$, and with a well-defined terminal state.

B. Q-Learning, ANNs, and NFQ

We will employ Neural Fitted Q Iteration (NFQ) [1] in order to train our agent. NFQ is closely related to traditional Q-learning combined with artificial neural networks (ANNs). Recall that in Q-learning, we are trying to learn a function that estimates the quality of a particular state-action pair, i.e:

$$Q: S \times A \rightarrow \mathbb{R}$$

In Q-learning without function approximation (i.e. with a lookup table), the action value function update is given by [7]

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha (r_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t))$$

However, when the state space is continuous or very large, it is not feasible to use a lookup table. Instead we must employ function approximation techniques to approximate the Q function. Artificial neural networks (ANNs), or multi-layer perceptrons are well suited to this task, since they are able to learn a wide variety of nonlinear functions. An example of an ANN architecture is presented in Figure 1. Now, instead of updating the Q value directly, we backpropagate errors via gradient descent, like the commonly used squared error measure

$$e = (Q(s_t, a_t) - (r_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)))^2$$

although we could choose another error measure if we so wish. Here, as in traditional Q-learning, the update happens on-line. This should lead to productive generalization such that portions of the state-action space that are similar to each other may map to similar Q values. While this will be true in the limit, unproductive over-generalization of the state-action space mapping may occur early in the learning process, slowing down learning.
This leads us to NFQ. Unlike Q learning, NFQ does not update on-line. Instead, NFQ first gathers data for some epoch, then updates off-line for the entire history of training experiences. NFQ uses the same update rule as Q learning, except without the learning rate $\alpha$. Because we are learning on our full history of experiences, none of our experience is forgotten. Thus, the NFQ update rule is given by

$$Q(s_t, a_t) \leftarrow r_t + \gamma \max_a Q(s_{t+1}, a)$$

There are several of advantages to NFQ over traditional Q-learning combined with ANNs when applied to our problem. Since NFQ is able to learn off-line, it can eschew on-line gradient descent techniques for faster and more reliable supervised learning methods. Following Riedmiller (2005), we will use Rprop, which is both fast and robust to the choice of parameters [2]. This reduces our parameter set so we may focus on tuning parameters related to the game itself (such as the choice of features) instead of trying to tease out the relative importance of the many available choices. More importantly, Riedmiller finds that NFQ helps to limit the over-generalization that may occur if the update step is applied after each new sample is collected, while still allowing for some generalization through the neural networks.

C. Previous Work

The application of machine learning methods to board games has a long history. In particular, many have attempted to learn games via reinforcement learning. For a quick sampling, consider NeuroChess in chess [5], Chinook in checkers [3], Go [4], Othello [6], among others. Perhaps the most notable contribution to this field is Gerard Tesauro’s TD-Gammon, which was able to learn to play at an extremely high level using an artificial neural network for its value function trained using TD($\lambda$). This paper attempts to build on these previous studies by (a) employing an algorithm that so far has seen little use in the learning of board games, and (b) applying the reinforcement learning approach to a class of games (Eurogames) so far unstudied.
II. Euphoria

In the past two decades, there has been a resurgence of interest in boardgames and board game design, and in particular in Euro-style games (or just Eurogames), so named because of the genre’s origins in Germany. Today, designers from all over the world have embraced the Eurogame paradigm due to its focus on deep strategy, novel and interesting mechanics, and the use of stochastic transitions only when mechanically interesting. Here, we focus on the game *Euphoria*, published in 2013 by Stonemeier Games, which in many ways is designed like a typical Eurogame, with the addition of occasionally stochastic reward assignment and state transition, and hidden information to make our problem more interesting.

*Euphoria* is a worker placement and resource management game. Its core mechanic is as follows: you have an allocation of workers (dice). On your turn, you may either

1. Place a worker on a location on the board (assuming you have workers left in reserve to place, and assuming you can pay any relevant costs), reaping the (possibly stochastic and/or state-dependent) rewards granted by that location, and “bumping” any worker already on this space back to its owner, OR

2. Retrieve your workers (if you have any workers on the board to retrieve), and roll them to generate their new values

Certain board locations grant additional workers, while others grant one or several of the seventeen resources. Most importantly, some locations allow to player to place stars, which are the game’s victory resource. The first player to place all of his or her stars wins the game.

The player must manage the dual constraints of both labor and physical resources consistent with the current distance to game end. The game also includes an interesting risk/reward mechanic regarding the number of workers held. Clearly, it is costly in game time to retrieve workers. However, workers in reserve have a collective knowledge represented by their rolled dice values. If the total of their values exceeds a known threshold, then a costly worker is lost. Full rules for the game can be found here: [http://stonemaiergames.com/rules-euphoria/](http://stonemaiergames.com/rules-euphoria/). Figure 2 shows a game of *Euphoria* in progress.

We aim to implement an agent for *Euphoria*. Resource, labor, and risk management while maximizing total reward are all problems that seem well suited to the MDP framework. Since the state space of the game is large, far too large to manage with a lookup table, we must represent the Q function with a neural network, and will update it using NFQ. Although the action set has a maximum size of 113 moves, the action set is highly state dependent, and generally only 10-15 actions will be available in a given state. This will be beneficial to us, since MDPs with large action sets may not converge, or may converge much more slowly without special consideration taken to manage the large action space.

III. Experimental Setup

We follow the model of TD-Gammon when creating our feature vector. We try to individually encode in binary form as many features as may be relevant to the success of the agent, with
features that would naturally be represented as integers instead represented as a series of binary variables. For example, consider the case when a player has two gold. We have allocated five features to this player’s gold stash, four for the first four gold collected, and a final feature that represents the number of additional gold held beyond four, scaled to a number between 0 and 1. In this case the first two digits of the extended feature would be turned on, while the other three would remain off. Tesauro finds that this style of feature encoding is more conducive to the use of neural networks than integer encodings. The other features are defined analogously, and the full feature vector is described below. The number in parentheses denotes the total number of features in that category.

- binary features for the presence of a given player’s worker on a given position on the board (98)
- binary features for each player’s worker stock, active and inactive (16)
- binary features for potentially relevant values of each of the 17 in-game resources, non-binary scaled spillover features for resources held beyond the binary thresholds (186)
- binary game-state features for the various allegiance tracks, mine statuses, lands left on board, market statuses (165)
- binary features for recruit factions, turn (4)

The feature vector has 469 total components.

The agent is given a score of +1 for winning a game, and −1 for losing the game. In the case of a tie, the agent receives a score of 0. The neural networks have a feed-forward
topology, a linear input for each feature, the same number of sigmoidal hidden layers, where
the sigmoidal function is given by

\[ f(x) = \frac{1}{1 + e^{-x}} \]

and a single linear output layer. We use \( \epsilon \)-greedy exploration for initial \( \epsilon = 0.3 \) and a decay
rate of 0.999. We use a discount factor of \( \gamma = 0.9 \), and omit \( \alpha \) for the reasons described
above. The training-testing procedure is as follows:

1. generate a game’s worth of data against an opponent who plays legally, but randomly
2. train using NFQ
3. test by playing 100 games against the random agent, take average score over those games

IV. Results and Conclusion

Figure 3 shows average testing performance as a function of total NFQ training epochs (full
games). Testing performance is denoted by average score over the training games. Recall
that the agent receives a reward of +1 if it wins, and a reward of −1 if it loses. Thus, a score
of 0 would represent winning and losing in equal amounts. Notice that the agent appears
to have learned a lot (relative to initial performance) in the first few epochs. After this
point, increases in performance are more erratic, although it does appear that the overall
trend is positive. It seems that after the first few training rounds, our NFQ agent is able to
consistently beat its opponent more than 50% of the time, with occasional jumps into the
75% range (although still below 50% on occasion).

There are two primary next steps in the training process. First, more training. The first
version of TD-Gammon trained for 300,000 games, and later versions trained for 1,500,000.
Forty training rounds is nothing compared to this. Second, self-training. It is well known
that training against random agents is woefully insufficient preparation if an agent wishes to
challenge a human opponent. Agents can become caught in “strategy traps” that can deftly
handle random agents, but fail to even the most naive strategies. However, for as few epochs
as this agent was trained, this is an encouraging result.

All code for this project can be found at https://github.com/myeaton1/euphoriaAI.
**Figure 3.** Average NFQ Performance as a Function of Training Epochs
References


