Machine Learning Nanodegree
Capstone Report

David A. Robles

January 31, 2017

1 Definition

1.1 Project Overview

Content.

1.2 Problem Statement

In this project, we will use reinforcement learning with deep learning to make an agent learn to play the game of Connect 4[1] by playing games against itself. In other words, using the formalism used by Mitchell (1997) to define a machine learning problem:

- **Task:** Playing Connect 4.
- **Performance:** Percent of games won against other agents, and accuracy of the predictions on a Connect 4 dataset.
- **Experience:** Games played against itself.
- **Target function:** $Q^\pi : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$, where $\mathcal{S}$ is the set of *states* (board positions) and $\mathcal{A}$ is the set of *actions* (moves), and $\mathbb{R}$ represents the value of being in a state $s \in \mathcal{S}$, applying a action $a \in \mathcal{A}$, and following policy π thereafter.
- **Target function representation:** Deep neural network.

Therefore, I seek to build a Q-learning agent trained via a deep convolutional neural network to approximate the optimal action-value function:

$$Q^*(s,a) = \max_\pi Q^\pi(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A} \tag{1}$$

which is the maximum sum of rewards achievable by a behaviour policy π.

1.3 Metrics

- **Winning percentage.** This metric consists in playing a high number of games (e.g. 100,000) against another agent (e.g. a random agent), and calculating the average of games won by the agent that uses the learned value function.
- **Prediction accuracy.** The learned value function will be used to predict the game-theoretic outcomes (win, loss or draw) of the board positions in the Connect 4 Data Set.

[1] https://en.wikipedia.org/wiki/Connect_Four

2 Analysis

2.1 Data Exploration

Content.

2.2 Exploratory Visualization

Content.

2.3 Algorithms and Techniques

2.3.1 Alpha-beta pruning

Implementation

Alpha-beta pruning (Knuth and Moore, 1975) is the most common game tree search algorithm for two-player games of perfect information. It extends the minimax algorithm to reduce the number of nodes that are evaluated in the game tree. Instead of calculating the exact minimax values for all the nodes in the game tree, alpha-beta prunes away branches that will not have any effect in the selection of the best move.

2.3.2 Q-learning

Implementation

One of the most basic and popular methods to estimate action-value functions is the *Q-learning* algorithm (Watkins, 1989). It is model-free online off-policy algorithm, whose main strength is that it is able to compare the expected utility of the available actions without requiring a model of the environment. Q-learning works by learning an action-value function that gives the expected utility of taking a given action in a given state and following a fixed policy thereafter. The update rule uses action-values and a built-in max-operator over the action-values of the next state in order to update $Q(s_t, a_t)$ as follows,

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)] \tag{2}$$

The agent makes a step in the environment from state s_t to s_{t+1} using action a_t while receiving reward r_t. The update takes place on the action-value a_t in the state s_t from which this action was executed.

Q-learning is exploration-intensive, which means that it will converge to the optimal policy regardless of the exploration policy being followed, under the assumption that each state-action pair is visited an infinite number of times, and the learning parameter α is decreased appropriately (Watkins and Peter, 1992).

2.3.3 Self-play

Self-play is by far the most popular training method. It is a single policy $\pi(s, a)$ that is used by both players in a two-player game, $\pi_1(s, a) = \pi_2(s, a) = \pi(s, a)$. The first reason for its popularity is that training is quickest if the learner's opponent is roughly equally strong, and that definitely holds for self-play. As a second reason for popularity, there is no need to implement or access a different agent with roughly equal playing strength. However, self-play has several drawbacks, with the main one being that a single opponent does not provide sufficient exploration (Szita, 2011).

2.4 Benchmark

- **Random agent**. This benchmark consists in playing against an agent that takes uniformly random moves. This is the most basic benchmark, but first we have to be sure that our learned evaluation function can play better than a random agent before moving into a harder benchmark. Also, this will help us to detect bugs in the code and algorithms: if a learned value function does not play significantly better than a random agent, is not learning. The idea is to test against this benchmark using Alpha-beta pruning at 1, 2 and 4-ply search.

- **Connect 4 Data Set**. This dataset will be used as the main benchmark. The learned value function will be used to predict the game-theoretic outcomes (win, loss or draw) for the first player in the 67,557 instances of the dataset.

3 Implementation

3.1 Markov Decision Process

Implementation

A *Markov decision process* (MDP) consist of a set of states, a set of actions, a transition function and a reward function.

- $\mathcal{S}$ is the set of *states* (state space).
- $\mathcal{A}$ is the set of *actions* (action space). The set of actions that are available in some particular state $s_t \in \mathcal{S}$ is denoted $\mathcal{A}(s_t)$, such that $\mathcal{A}(s_t) \in \mathcal{P}(\mathcal{A})$, where $\mathcal{P}(\cdot)$ denotes the power set.
- $T : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ is the *transition function*, which is the probability given we are in state $s_t \in \mathcal{S}$, take action $a_t \in \mathcal{A}(s_t)$ and we will transition to state $s_{t+1} \in \mathcal{S}$.
- $R : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ is the *reward function*, which is the immediate reward received when in state $s_t \in \mathcal{S}$ action $a_t \in \mathcal{A}$ is taken and the MDP transitions to state $s_{t+1} \in \mathcal{S}$. However, it is also possible to define it either as $R : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ or $R : \mathcal{S} \to \mathbb{R}$. The first one gives rewards for performing an action a_t in a particular state s_t, and the second gives rewards when transitioning to state s_{t+1}.

3.2 Environment

Implementation

An agent does not have access to the dynamics (reward and transition function) of the MDP. However, it interacts with an *environment* by way of three signals: a *state*, which describes the state of the environment, an *action*, which allows the agent to have some impact on the environment, and a *reward*, which provides the agent with feedback on its immediate performance.

3.3 Policy

Implementation

In an MDP, the agent acts according to a policy π, which maps each state $s \in \mathcal{S}$ to an action $a \in \mathcal{A}(s)$. A policy that specifies a unique action to be performed is called a *deterministic* policy, and is defined as $\pi : \mathcal{S} \to \mathcal{A}$. On the other hand, a *stochastic* policy $\pi : \mathcal{S} \times \mathcal{A} \to [0, 1]$ selects actions according to a probability distribution, such that for each state $s \in \mathcal{S}$, it holds that $\pi(s, a) \geq 0$ and $\sum_{a \in \mathcal{A}(s)} \pi(s, a) = 1$.

The interaction between the policy used by the agent and the environment works as follows. First, it starts at an *initial state* s_0. Then, the policy π selects an action $a_0 = \pi(s_0)$ from the set of available actions $\mathcal{A}(s_0)$, and the action is executed. The environment transitions to a new state s_1 based on the transition function T with probability $T(s_0, a_0, s_1)$, and a reward $r_0 = R(s_0, a_0, s_1)$ is received. This process continues, producing a *trajectory* of experience $s_0, a_0, s_1, r_1, a_1, s_2, r_2, a_2, \ldots$. If the task is *episodic*, the process ends in a *terminal state* s_T and is restarted in the initial state. If the task is *continiuing*, the sequence of states can be extended indefinitely.

3.3.1 Random policy

Implementation

Selects:

$$\pi_{\text{rand}}(s) = \operatorname*{argmax}_{a \in \mathcal{A}(s_t)} Q(s, a) \tag{3}$$

3.3.2 Greedy policy

Implementation

The greedy tree policy selects the *max action*, which is the greedy action with the highest value.

$$\pi_{\text{greedy}}(s) = \operatorname*{argmax}_{a \in \mathcal{A}(s_t)} Q(s, a) \tag{4}$$

3.3.3 ϵ-greedy

Implementation

The ϵ-greedy tree policy selects the best action for a proportion $1 - \epsilon$ of the trials, and another action is randomly selected (with uniform probability) for a proportion ϵ,

$$\pi_{\epsilon}(s) = \begin{cases} \pi_{\text{rand}}(s, a) & \text{if } rand() < \epsilon \\ \pi_{\text{greedy}}(s, a) & \text{otherwise} \end{cases} \tag{5}$$

where $\epsilon \in [0, 1]$ and $rand()$ returns a random number from a uniform distribution $\in [0, 1]$.

4 Methodology

4.1 Data Preprocessing

Content.

4.2 Implementation

- Two games were implemented: Tic Tac Toe and Connect 4.
- Converting a to an MDP by using a fixed agent.
- Show that q-learning learns to play against a fixed opponent using Tic-Tac-Toe.
- Show that q-learning learns to play against a fixed opponent using Connect 4.

4.3 Refinement

Content.

4.4 Games

4.4.1 Tic Tac Toe

Implementation

4.4.2 Connect 4

Implementation

Connect 4 is a two-player board game of perfect information where pieces are dropped into the columns of a vertical 6×7 grid with the goal of forming a straight line of 4 connected pieces. There are at most 7 actions per state, since placing a piece in a column is a legal action only if that column has at least one empty location. Figure 1 shows three Connect 4 game positions.

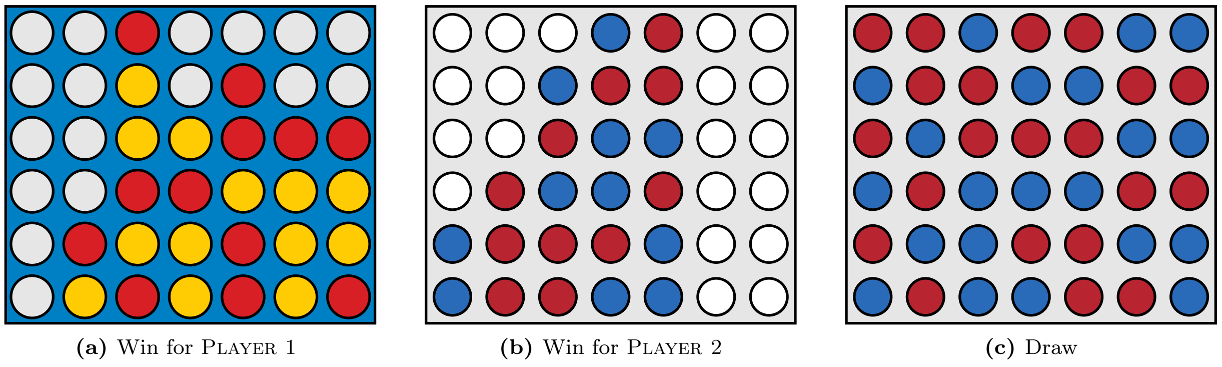

(a) Win for Player 1 **(b)** Win for Player 2 **(c)** Draw

Figure 1: Examples of wins, losses and draws in Connect Four

4.4.3 GameMDP converter

Converting an MDP

4.4.4 Learn Tic-Tac-Toe state-action values using Q-learning

Implementation

First, we will test our q-learning algorithm by learning the state action values for an agent that plays against a fixed tic tac toe player. this fixed tic tac toe player will be an AlphaBeta player, which means it will always make an optimal move. An optimal move means that is the best possible move in that situation.

We created a custom board position that is small but that it can demonstrate the idea of learning the q-learning values.

The situation consists where we are in the following position:

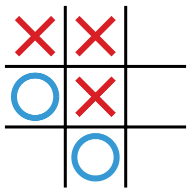
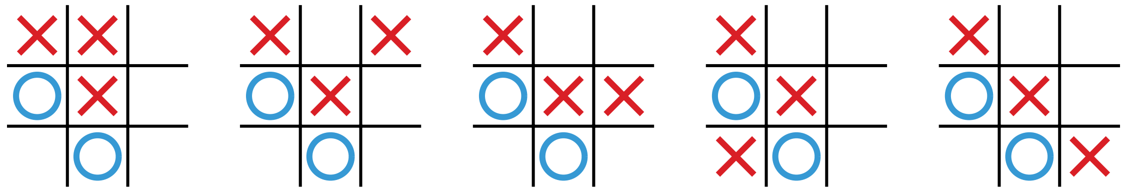


Figure 2: Tic-Tac-Toe board, s, with five legal moves: $\{2, 3, 6, 7, 9\}$.

In this position the player X is next, and it has 5 available moves: 1, 3, 6, 9.

We ran the Q-learning algorithm with the following parameters and these are the results:

As we can see we were able to learn the state-action values for each board position. It is clear that the best board is the one that leads to a direct win, and the other boards lead to either draws (like in...) or a loss.



(a) $Q(s,2) = 0.9895$ **(b)** $Q(s,3) = 0.9897$ **(c)** $Q(s,6) = 0.0$ **(d)** $Q(s,7) = 0.9890$ **(e)** $Q(s,9) = 0.9999$

Figure 3

Now we will run the same algorithm but starting from the initial position.

As we can see it learned to play perfectly against an alpha-beta player.

With this example we can confirm that the implementation of our q-learning algorithm is working fine, and now we can move into our next step.

5 Results

5.1 Model Evaluation and Validation

Content.

5.2 Justification

Content.

6 Conclusion

6.1 Free-Form Visualization

Content.

6.2 Reflection

Content.

6.3 Improvement

Content.

References

Donald E. Knuth and Ronald W. Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6(4), 1975.

T. M. Mitchell. *Machine Learning*. McGraw Hill, 1997.

I. Szita. Reinforcement learning and markov decision processes. In *Reinforcement Learning: State of the Art*. Springer, 2011.

Christopher J.C.H. Watkins and Dayan Peter. Q-learning. *Machine Learning*, 8:279–292, 1992.

C.J.C.H. Watkins. *Learning from Delayed Rewards*. PhD thesis, Cambridge University, 1989.