Learning to Play Board Games With Reinforcement Learning

David A. Robles

May 4, 2017

1 Definition

1.1 Project Overview

Reinforcement learning is the area of machine learning concerned with the idea of learning by interacting with an environment (Sutton and Barto, 1998). It has a long and notable history in games. The checkers program written by Samuel (1959) was the first remarkable application of temporal difference learning, and also the first significant learning program of any kind. It had the principles of temporal difference learning decades before it was described and analyzed. However, it was another game where reinforcement learning reached its first big success, when TD-Gammon (Tesauro, 1995) reached world-class gameplay in Backgammon by training a neural network-based evaluation function through self-play.

Deep Learning (LeCun et al., 2015) is another branch of machine learning that allows computational models that are composed of multiple processing layers to learn representations of data with multiple levels of abstraction. Deep learning techniques have dramatically improved the state-of-the-art in areas such as speech recognition (Hinton et al., 2012), image recognition (Krizhevsky et al., 2012) and natural language processing (Colbert et al., 2011).

Reecently, there have been several breakthroughs when combining reinforcement learning and deep learning. Mni et al. (2015) used a convolutional neural network trained with a variant of Q-learning and learned to play Atari 2600 games at a human-level. Last year, one of the biggest challenges for artificial intelligence was solved, when Google's DeepMind (Silver et al., 2016) created AlphaGo to beat the world's greatest Go player. AlphaGo used deep neural networks trained by a combination of supervised learning from human expert games, and reinforcement learning from games of self-play.

Developing strong game-playing programs for classic two-player games (e.g. Chess, Checkers, Go) is important in two respects: first, for humans that play the games looking for an intellectual challenge, and second, for AI researchers that use the games as testbeds for artificial intelligence. In both cases, the task for game programmers and AI researchers of writing strong game AI is a hard and tedious task that requires hours of trial and error adjustments and human expertise. Therefore, when a strong game-playing algorithm is created to play a specific game, it is rarely useful for creating an algorithm to play another game, since the domain knowledge is non-transferable. For this reason, there is enormous value in using machine learning to learn value functions to play these games without using any domain knowledge.

1.2 Problem Statement

In this project we use reinforcement learning and deep learning to learn value functions for the games of Tic-Tac-Toe and Connect 4. We define the machine learning problem as follows:

- **Task:** Playing Tic-Tac-Toe and Connect 4.
- **Performance:** Winning percentage when playing against a random player, starting from new games, and also starting from board positions from the UCI Connect 4 dataset.
- **Experience:** Games played against a random opponent, an Alpha-Beta opponent and against itself.

- **Target function:** $Q^{\pi} : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$, 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 representations:** Lookup table and deep neural network.

More specifically, we seek to build an agent that uses Q-learning via self-play to train 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 behavior policy π.

1.3 Metrics

- **Winning percentage.** Consists in playing a high number of games (e.g. 100,000) against other agents (e.g. Random player) using the learned value functions as the action-value function to take greedy actions.

2 Analysis

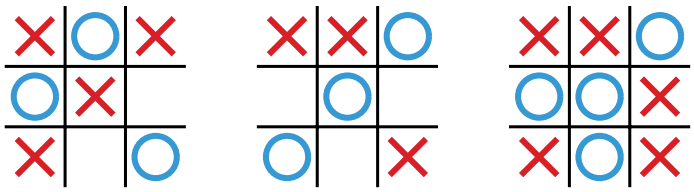
2.1 Data Exploration and Visualization

In this project we use two environments and one dataset. The environments are the games of Tic-Tac-Toe and Connect 4, and the dataset is the UCI Connect 4 dataset.

2.1.1 Tic-Tac-Toe Environment

Tic-Tac-Toe [implementation] is a paper-and-pencil game for two players, X and O, who take turns marking the spaces in a 3×3 grid. The player who succeeds in placing three of their marks in a horizontal, vertical, or diagonal row wins the game. Figure 1 shows three Tic-Tac-Toe game positions: a win, a draw and a loss.

Tic-Tac-Toe is an extremely simple game, nonetheless, is very useful to analyze simple concepts and to verify that the implementation of the learning algorithms are behaving as expected before moving into the Connect 4 environment, which is more complex because of its large state space.

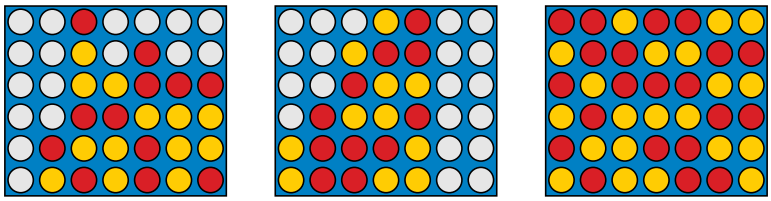


(a) Win for X **(b)** Win for O **(c)** Draw

Figure 1: Tic-Tac-Toe

2.1.2 Connect 4 Environment

Connect 4 [implementation] 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 seven actions per state, since placing a piece in a column is a legal action only if that column has at least one empty location. In this project we use pieces of two colors: YELLOW for the first player, and RED for the second player. Figure 2 shows three Connect 4 game positions: a win for YELLOW, a win for RED and a draw:



(a) Win for YELLOW **(b)** Win for RED **(c)** Draw

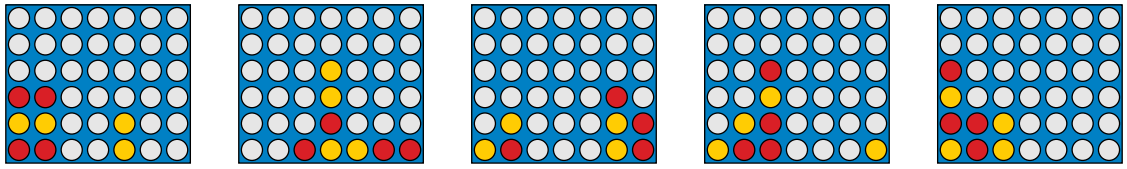
Figure 2: Connect Four

2.1.3 UCI Connect 4 Data Set

As part of the testing phase, we will use the *Connect 4 Data Set* that is available from the UCI Machine Learning Repository (Lichman, 2013). A partial view of the dataset is presented in Table 1. The dataset has a total of 67,557 instances, representing all legal 8-ply positions in Connect 4 in which neither player has won yet, and which the next move is not forced. Each instance is described by 42 features, one for each space in the 6×7 board, and can belong to one of the classes $\{\texttt{x}, \texttt{o}, \texttt{b}\}$, where `x` is the first player, `o` is the second player, and `b` is empty. The outcome class is the game theoretical value for the first player, and can belong to one of the classes $\{\texttt{win}, \texttt{loss}, \texttt{draw}\}$. There are 44,473 wins, 16,635 losses and 6,449 draws [implementation]. Figure 3 shows a visual representation of five randomly selected instances of the data set. As we can see, all game positions have eight pieces in the board, representing all legal 8-ply positions [implementation].

	Features																	Target
No.	**a1**	**a2**	**a3**	**a4**	**a5**	**a6**	**b1**	**b2**	**...**	**f5**	**f6**	**g1**	**g2**	**g3**	**g4**	**g5**	**g6**	**outcome**
0	b	b	b	b	b	b	b	b	...	b	b	b	b	b	b	b	b	win
1	b	b	b	b	b	b	b	b	...	b	b	b	b	b	b	b	b	win
2	b	b	b	b	b	b	o	b	...	b	b	b	b	b	b	b	b	win
3	b	b	b	b	b	b	b	b	...	b	b	b	b	b	b	b	b	win
4	o	b	b	b	b	b	b	b	...	b	b	b	b	b	b	b	b	win
...	..	..	..	..	..	..	..	..	...	..	..	..	..	..	..	..	..	...
67552	x	x	b	b	b	b	o	x	...	b	b	o	o	x	b	b	b	loss
67553	x	x	b	b	b	b	o	b	...	b	b	o	x	o	o	x	b	draw
67554	x	x	b	b	b	b	o	o	...	b	b	o	x	x	o	b	b	loss
67555	x	o	b	b	b	b	o	b	...	b	b	o	x	o	x	x	b	draw
67556	x	o	o	o	x	b	o	b	...	b	b	x	b	b	b	b	b	draw

Table 1: UCI Connect 4 Dataset. Each row represents a different board position, and each feature represents a specific cell in the board. The target is the outcome of the game for the first player, assuming perfect play.



(a) Win **(b)** Loss **(c)** Win **(d)** Loss **(e)** Draw

Figure 3: Five randomly selected instances of the UCI Connect 4 Data Set. The outcome of each board position is from the point of view of the firs player (yellow discs).

2.2 Algorithms and Techniques

2.2.1 Markov Decision Process

A *Markov decision process* (MDP) [implementation] consist of four elements:

- $\mathcal{S}$ 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 $\mathcal{A}(s_t)$.
- $T : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ 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}.

2.2.2 Environment

In the reinforcement learning problem an agent does not have access to the dynamics (reward and transition functions) of the MDP. However, it interacts with an *environment* [implementation] 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.

2.2.3 Policy

In an MDP, the agent acts according to a policy π [implementation], 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 $\pi : \mathcal{S} \to \mathcal{A}$.

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 $s_0, a_0, s_1, r_1, a_1, s_2, r_2, a_2, \ldots$, and the process ends in a *terminal state* s_T and is restarted in the initial state.

We use three types of policies in this project:

- **Random.** Selects actions uniformly at random [implementation].
- **Greedy.** Selects the *max action*, which is the greedy action with the highest value [implementation],

$$\pi_{\text{greedy}}(s) = \underset{a \in \mathcal{A}(s)}{\text{argmax}}\, Q(s, a) \tag{2}$$

- **ϵ-greedy.** Selects the best action for a proportion $1 - \epsilon$ of the trials, and another action is randomly selected (with uniform probability) for a proportion [implementation],

$$\pi_\epsilon(s) = \begin{cases} \pi_{\text{rand}}(s, a) & \text{if } rand() < \epsilon \\ \pi_{\text{greedy}}(s, a) & \text{otherwise} \end{cases} \tag{3}$$

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

2.2.4 Value Functions

Most of the algorithms for solving MDPs (computing optimal policies) do it by learning a *value function* [implementation]. A value function estimates what is good for an agent over the long run. It estimates the expected outcome from any given state, by summarizing the total amount of reward that an agent can expect to accumulate into a single number. Value functions are defined for particular policies.

The *state value function* (or V-function), is the expected return when starting in state s and following policy π thereafter (Sutton and Barto, 1998),

$$V^\pi(s) = \mathbb{E}_\pi \left[R_t | s_t = s \right] \tag{4}$$

The *action value function* (or Q-function), is the expected return after selecting action a in state s and then following policy π,

$$Q^\pi(s, a) = \mathbb{E}_\pi \left[R_t | s_t = s, a_t = a \right] \tag{5}$$

The *optimal value function* is the unique value function that maximises the value of every state, or state-action pair,

$$Q^*(s, a) \quad = \quad \max_\pi Q^\pi(s, a), \forall s \in \mathcal{S}, a \in \mathcal{A} \tag{6}$$

An *optimal policy* $\pi^*(s, a)$ is a policy that maximises the action value function from every state in the MDP,

$$\pi^*(s, a) = \underset{\pi}{\text{argmax}}\, Q^\pi(s, a) \tag{7}$$

2.2.5 Q-learning

One of the most basic and popular methods to estimate action-value functions is the *Q-learning* algorithm (Watkins, 1989) [implementation]. 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,

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right] \tag{8}$$

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. This version of Q-learning works well for tasks with a small a state-space, since it uses arrays or tables [implementation] with one entry for each state-action pair.

2.2.6 Approximate Q-learning

In many cases in which there are far more states than could possibly be entries in a table we need to use function approximation. Approximate Q-learning [implementation] consists in parameterizing an approximate action-value function, $Q(s, a; \theta_i) \approx Q(s, a)$, in which θ_i are the parameters (weights) of the action-value function at iteration i. Usually the number of parameters of a function approximator is much less than the state space, which means that a change in one parameter can change many Q-values, as opposed to just one as in the tabular case.

2.2.7 Experience Replay

Reinforcement learning is known to be unstable or even to diverge when a nonlinear function approximator such as a neural network is used to represent the action-value function. One trick to make it work is to use *experience replay* (Lin, 1993) [implementation], which consists in storing the experiences (s_t, a_t, r_t, s_{t+1}) at each time step t in a data set $D_t = \{e_1, \ldots, e_t\}$. During the training of approximate Q-learning random minibatches from the replay memory are used instead of the most recent transition. This breaks the similarity of subsequent training samples, which otherwise might drive the network into a local minimum. The Q-learning update at iteration i uses the following loss function (Mni et al., 2015):

$$L_i(\theta_i) = \mathbb{E}_{(s,a,r,s') \sim U(D)} \left[\left(r + \gamma \max_{a'} Q(s', a', \theta_i^-) - Q(s, a; \theta) \right)^2 \right] \tag{9}$$

where $(s, a, r, s') \sim U(D)$ is a sample minibatch of experience drawn uniformly at random from the memory pool of stores experiences.

2.2.8 Self-play

Self-play [implementation] is by far the most popular training method for games. 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).

2.2.9 Convolutional Neural Networks

Convolutional Neural Networks (LeCun, 1989), or CNNs, are a special type of neural network that has a known grid-like topology. Like most other neural networks they are trained with a variant of the backpropagation algorithm. CNNs strengh is pattern recognition directly from pixels of images with minimal processing. We use a convolutional network as a function approximator for the board of Connect 4 [implementation], since the board can be thought of as a 2-D grid of pixels (discs in this case).

2.3 Benchmark

- **Random agent**. This benchmark consists in playing against an agent that takes uniformly random moves. 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.
- **Connect 4 Data Set**. The board configurations in this dataset will be used as starting positions to play games against the random agent. Starting from positions where we known the game-theoretic outcome is helpful to know how good the learning agent can get.

3 Methodology

3.1 Data Preprocessing

The only data preprocessing done in this project was for the Connect 4 UCI dataset. As mentioned before, the dataset was used only for the testing phase. All the learning came from the two reinforcement learning simulators for Tic-Tac-Toe and Connect 4. We loaded the Connect 4 UCI dataset into a Pandas DataFrame for simple data analysis using `load_dataframe` [implementation]. Also, a `series_to_game` helper function was created to convert a DataFrame row into a Connect 4 game [implementation], and another three helper functions were created to load games randomly, or by specific game-theoretic outcome [implementation].

3.2 Implementation

3.2.1 Game

The first part of the implementation was to define a `Game` interface [implementation] to be used by both games: Tic-Tac-Toe and Connect 4.

```
class Game(object):

    def copy(self):
        '''Returns a copy of the game.'''
        pass

    def cur_player(self):
        '''
        Returns the index of the player in turn, starting with 0:
        0 (Player 1), 1 (Player 2), etc.
        '''
        pass

    def is_over(self):
        '''Returns True if the game is over.'''
        return len(self.legal_moves()) == 0

    def legal_moves(self):
        '''Returns a list of legal moves for the player in turn.'''
        pass

    def make_move(self, move):
        '''Makes one move for the player in turn.'''
        pass

    def outcomes(self):
        '''Returns a list of outcomes for each player at the end of the game.'''
        pass

    def reset(self):
        '''Restarts the game.'''
        pass
```

Once we had the `Game` interface well defined we implemented Tic-Tac-toe [implementation] and Connect 4 [implementation].

3.2.2 Markov Decision Process

Next, the interface for an MDP [implementation]:

```
class MDP(object):

    def states(self):
        '''Returns a list of all states. Not generally possible for large MDPs.'''
        pass

    def start_state(self):
        '''Returns the initial state.'''
        pass

    def actions(self, state):
        '''Returns a list of possible actions in the given state.'''
        pass

    def transitions(self, state, action):
        '''
        Returns a dict of (next_state: probability) key/values, where 'next_state' is
        reachable from 'state' by taking 'action'. The sum of all probabilities should
        be 1.0. Not available in reinforcement learning.
        '''
        pass

    def reward(self, state, action, next_state):
        '''
        Returns the reward of being in 'state', taking 'action', and ending up
        in 'next_state'. Not available in reinforcement learning.
        '''
        pass

    def is_terminal(self, state):
        '''.Returns True if the given state is terminal.'''
        pass
```

3.2.3 Markov Decision Process for Games

We also created two helper classes. The first class `GameMDP` [implementation] was designed to convert a Game into an MDP. In this case the agent makes moves for both players in the game. Used in self-play learning.

```
class GameMDP(MDP):

    def __init__(self, game):
```

```
        self._game = game
        self._states = {}

    def actions(self, state):
        return [None] if state.is_over() else state.legal_moves()

    def is_terminal(self, state):
        return state.is_over()

    def reward(self, state, action, next_state):
        '''Returns the utility from the point of view of the first player.'''
        return utility(next_state, 0) if next_state.is_over() else 0

    def start_state(self):
        return self._game.copy()

    def states(self):
        if not self._states:
            def generate_states(game):
                '''Generates all the states for the game'''
                if game not in self._states:
                    self._states[game] = game
                for move in game.legal_moves():
                    new_game = game.copy().make_move(move)
                    generate_states(new_game)
            generate_states(self._game)
        return self._states

    def transitions(self, state, action):
        if state.is_over():
            return [(state, 1.0)]
        new_game = state.copy().make_move(action)
        return [(new_game, 1.0)]
```

And also, `FixedGameMDP` [implementation], which converts a `Game` into an `MDP` by using a fixed opponent for one of the players in the game:

```
class FixedGameMDP(GameMDP):

    def __init__(self, game, opp_player, opp_idx):
        '''
        opp_player: the opponent player
        opp_idx: the idx of the opponent player in the game
        '''
        super(FixedGameMDP, self).__init__(game)
        self._opp_player = opp_player
        self._opp_idx = opp_idx
        self._agent_idx = opp_idx ^ 1
```

```
    def reward(self, game, move, next_game):
        return utility(next_game, self._agent_idx) if next_game.is_over() else 0

    def start_state(self):
        new_game = self._game.copy()
        if not new_game.is_over() and new_game.cur_player() == self._opp_idx:
            chosen_move = self._opp_player.choose_move(new_game)
            new_game.make_move(chosen_move)
        return new_game

    def transitions(self, game, move):
        if game.is_over():
            return []
        new_game = game.copy().make_move(move)
        if not new_game.is_over() and new_game.cur_player() == self._opp_idx:
            chosen_move = self._opp_player.choose_move(new_game)
            new_game.make_move(chosen_move)
        return [(new_game, 1.0)]
```

3.2.4 Environment

A reinforcement learning environment [implementation] serves as a middleman between the policies and the MDP. The main idea is that when a learning agent interacts with an environment it does not have access to the transition and reward functions. It wraps the MDP.

```
class Environment(object):

    def __init__(self, mdp):
        self._mdp = mdp
        self._cur_state = self._mdp.start_state()

    def actions(self, state):
        '''Returns the available actions in the given state.'''
        return self._mdp.actions(state)

    def cur_state(self):
        '''Returns the current state.'''
        return self._cur_state.copy()

    def do_action(self, action):
        '''
        Performs the given action in the current state.
        Returns (reward, next_state).
        '''
        prev = self.cur_state()
        transitions = self._mdp.transitions(self.cur_state(), action)
        for next_state, prob in transitions:
            self._cur_state = next_state
        reward = self._mdp.reward(prev, action, self.cur_state())
```

```
        return reward, self.cur_state()

    def is_terminal(self):
        return self._mdp.is_terminal(self.cur_state())

    def reset(self):
        '''Resets the current state to the start state.'''
        self._cur_state = self._mdp.start_state()
```

3.2.5 EpisodicLearnerMixin

Q-learning and Approximate Q-learning algorithms are episodic learners. Their implementation includes the mixin `EpisodicLearnerMixin` to reuse functionality for running episodes and calling the callbacks that are used for generating plots and evaluations.

```
class EpisodicLearnerMixin(object):
    '''
    Mixin for learning value functions by interacting with an
    environment in an episodic setting.
    '''

    @property
    def value_function(self):
        if hasattr(self, 'vfunction'):
            return self.vfunction
        if hasattr(self, 'qfunction'):
            return self.qfunction

    def train(self, n_episodes, callbacks=None):
        '''Trains the model for a fixed number of episodes.'''
        callbacks = CallbackList(callbacks)
        callbacks.on_train_begin()
        for episode in range(n_episodes):
            callbacks.on_episode_begin(episode, self.value_function)
            self.env.reset()
            self.episode()
            callbacks.on_episode_end(episode, self.value_function)
        callbacks.on_train_end(self.value_function)
```

3.2.6 Tabular Value Function

We created a generic tabular value function [implementation] that can be used for both state $V(s)$ and state-action $Q(s, a)$ values. In many algorithms such as in Q-learning is helpful to initialize the values in the lookup table to random values when they do not exist. This is supported with the `init` parameter in the constructor.

```
```

```
_MEAN = 0.0
_STD = 0.3

class TabularVF(ValueFunction):

    def __init__(self, init=True, random_state=None):
        self.init = init
        self.random_state = check_random_state(random_state)
        self._table = {}

    def __setitem__(self, key, value):
        self._table[key] = value

    def __getitem__(self, key):
        if key not in self._table:
            if self.init:
                self._table[key] = self.random_state.normal(_MEAN, _STD)
            else:
                self._table[key] = 0
        return self._table[key]
```

3.2.7 Tabular Q-learning

Tabular Q-learning [implementation] includes the `EpisodicLearnerMixin` to reuse the code for learning in episodes. The constructor allows the `selfplay` parameter, which is used with a self-play policy to maximize/minimize the best Q-value.

```
class QLearning(EpisodicLearnerMixin):

    def __init__(self, env, policy, qfunction, learning_rate=0.1,
                 discount_factor=1.0, selfplay=False):
        self.env = env
        self.policy = policy
        self.qfunction = qfunction
        self.learning_rate = learning_rate
        self.discount_factor = discount_factor
        self.selfplay = selfplay

    def best_qvalue(self, state):
        if self.selfplay:
            best_func = max_qvalue if state.cur_player() == 0 else min_qvalue
            return best_func(state, self.env.get_actions(state), self.qfunction)
        return max_qvalue(state, self.env.get_actions(state), self.qfunction)

    def episode(self):
        while not self.env.is_terminal():
            state = self.env.get_state()
            action = self.policy.get_action(state)
            reward, next_state = self.env.do_action(action)
```

```
            best_qvalue = self.best_qvalue(next_state)
            target = reward + (self.discount_factor * best_qvalue)
            td_error = target - self.qfunction[state, action]
            self.qfunction[state, action] += self.learning_rate * td_error
```

3.2.8 Approximate Q-learning

The implementation of Approximate Q-learning [implementation] is very similar to the one for tabular Q-learning. The main difference is that it supports experience replay. When experience replay is enabled it can be configured by specifying the `batch_size` and the `replay_memory_size`.

```
class ApproximateQLearning(EpisodicLearnerMixin):

    def __init__(self, env, policy, qfunction, discount_factor=1.0, selfplay=False,
                 experience_replay=True, batch_size=32, replay_memory_size=10000):
        self.env = env
        self.policy = policy
        self.qfunction = qfunction
        self.discount_factor = discount_factor
        self.selfplay = selfplay
        self.experience_replay = experience_replay
        self.batch_size = batch_size
        self.replay_memory_size = replay_memory_size
        if self.experience_replay:
            self.replay_memory = deque(maxlen=self.replay_memory_size)

    def best_qvalue(self, state):
        func = None
        if self.selfplay:
            func = np.max if state.cur_player() == 0 else np.min
        else:
            func = np.max
        return self.qfunction.best_value(state, self.env.actions(state), func)

    def episode(self):
        while not self.env.is_terminal():
            state = self.env.get_state()
            action = self.policy.get_action(state)
            reward, next_state = self.env.do_action(action)
            if self.experience_replay:
                experience = (state, action, reward, next_state)
                self.replay_memory.append(experience)
                batch_size = min(len(self.replay_memory), self.batch_size)
                experiences = random.sample(self.replay_memory, batch_size)
                self.qfunction.minibatch_update(experiences)
            else:
                best_qvalue = self.best_qvalue(next_state)
                update = reward + (self.discount_factor * best_qvalue)
                self.qfunction.update(state, action, update)
```

3.3 Connect 4 Convolutional Neural Network

We implemented the convolutional neural network for Connect 4 [implementation] using Keras (Chollet et al., 2015). The Keras model is compiled in the constructor, and it takes the board as an input by vectorizing the board as:

```
def mapper(t):
    if t == 'X':
        return np.array([1.0])
    elif t == 'O':
        return np.array([−1.0])
    else:
        return np.array([0.0])
x = np.vectorize(mapper)(np.array(board))
x = x.reshape(C4.ROWS, C4.COLS, −1)
```

After the input layer, we used convolutions with 4 layers of 64 filters of size 3×3 and rectified linear non-linearities. In addition to the convolutions, we added a densely-connected layer with 256 units. Finally, the output layer has only one unit that fires the value of the given board with a *tanh* activation function to keep it in the range $[-1, +1]$.

```
    def __init__(self):
        model = Sequential()
        model.add(Conv2D(64, kernel_size=(3, 3), input_shape=(C4.ROWS, C4.COLS, 1)))
        model.add(Activation('relu'))
        model.add(Conv2D(64, kernel_size=(3, 3), padding='same'))
        model.add(Activation('relu'))
        model.add(Conv2D(64, kernel_size=(3, 3), padding='same'))
        model.add(Activation('relu'))
        model.add(Conv2D(64, kernel_size=(3, 3), padding='same'))
        model.add(Activation('relu'))
        model.add(Flatten())
        model.add(Dense(256))
        model.add(Dense(1))
        model.add(Activation('tanh'))
        model.compile(loss='mse', optimizer=SGD(lr=0.001))
        self.model = model
```

The convolutional netowork allows minibatch updates from the experience memory pool used in the Q-learning algorithm:

```
    def minibatch_update(self, experiences):
        xlist = []
```

```
        ylist = []
        for _, _, reward, next_state in experiences:
            x = normalize_board(next_state.board)
            if next_state.is_over():
                update = reward
            else:
                output = self.model.predict(np.array([x]))
                update = reward + (0.99 * output[0][0])
            y = np.array([update])
            xlist.append(x)
            ylist.append(y)
        x = np.array(xlist)
        y = np.array(ylist)
        self.model.train_on_batch(x, y)
```

4 Experiments and Results

4.1 Model Evaluation and Validation

4.1.1 Estimate Tic-Tac-Toe Q-values using tabular Q-learning versus fixed opponent

The first experiment [implementation] consisted in using tabular Q-learning to estimate the Q-values of a simple Tic-Tac-Toe board position, shown in Figure 4. We modeled this scenario as an MDP in which a move by nature is made for the second player using a fixed Alpha-Beta agent that makes optimal moves. The position is set as the initial state, s_0, and it has five available actions, $\mathcal{A}(s_0) = \{1, 2, 4, 6, 9\}$. This position is very close to the end of the game, which allows us to develop intuition of the expected Q-values from the game-theoretic point of view. For instance, we can clearly identify that the best available action is 9, which leads to an eventual win for the first player assuming optimal actions thereafter. On the other hand, any other action (i.e. 1, 2, 4 or 6) would lead to a loss, assuming perfect play from the opponent. The rewards given by the MDP are +1 for a win for the first player, −1 for a win for the second player, and 0 for a draw.

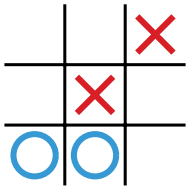
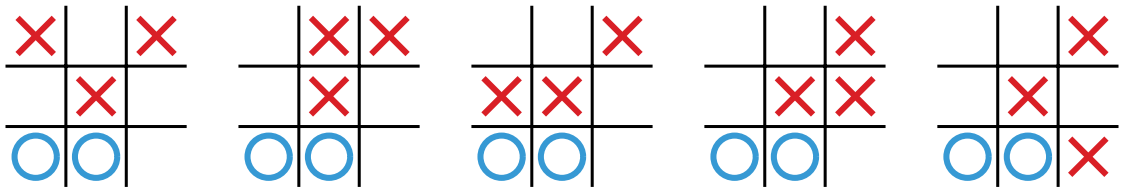


Figure 4: Initial state

We ran the experiment with $\alpha = 0.1$ (learning rate), $\gamma = 1.0$ (discount factor), and a π_{rand} (uniformly random) behavior policy. After approximately 800 episodes of training the Q-values converged to the expected, presented in Figure 5. As we can see, the expected future rewards are: 1.0 for action 9, and −1.0 for all other actions (i.e. 1, 2, 4 or 6). These are exactly the values we were looking for, which suggests that our implementation of Q-learning is correct.



(a) $Q(s_0, 1) = -1.0$ **(b)** $Q(s_0, 2) = -1.0$ **(c)** $Q(s_0, 4) = -1.0$ **(d)** $Q(s_0, 6) = -1.0$ **(e)** $Q(s_0, 9) = 1.0$

Figure 5

Figure 6 shows the predicted state-action values during training. It is interesting to see how the value of $Q(s_0, 9)$ takes more time to converge than the other Q-values. This is probably because taking action 9 results in an eventual, but not immediate win, and it requires making another move (either 1 or 6) to win. Also, is worth mentioning why we used a π_{rand} as the behavior policy instead of something like π_ϵ. Q-learning is an off-policy algorithm, which means we can use one policy for sampling (behavor policy), while estimating the value function for another one (target policy). And in an MDP as small as this, a simple π_{rand} policy is effective.

4.1.2 Estimate Tic-Tac-Toe Q-values using tabular Q-learning via self-play

In this experiment [implementation] we also used tabular Q-learning to estimate the Q-values of the same position described in Section 4.1.1. This time, however, we used self-play, which consists in playing games

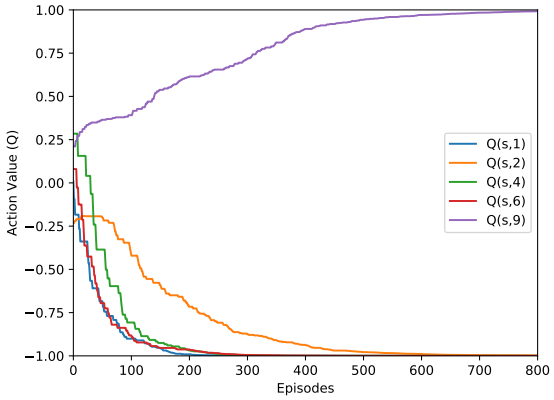


Figure 6: The predicted Q-values during the training phase of tabular Q-learning using Alpha-Beta as a fixed opponent.

against itself. Self-play is a good alternative when we do not have a strong opponent or one of roughly equal playing strength to train against. This is the case in Connect 4 because we cannot use the Alpha-Beta algorithm to search the full tree, given its large state-space and game tree complexity. To use Alpha-Beta we would need to use a heuristic evaluation function, which is what we are trying to create with deep reinforcement learning anyway. Nevertheless, we are trying to validate that our self-play implementation is working in Tic-Tac-Toe before moving to Connect 4.

We ran this experiment using the same parameters: $\alpha = 0.1$, $\gamma = 1.0$, and a π_{rand} behavior policy. Once again, the Q-values converged to the expected: 1.0 for action `9`, and -1.0 for all other actions. However, in Figure 7 we can observe how this time the Q-values took 3,500 episodes to converge, which is around four times slower than when training against a strong fixed opponent. This is understandable, since the opponent was itself, i.e., an agent that makes uniformly random moves and provides less direct feedback.

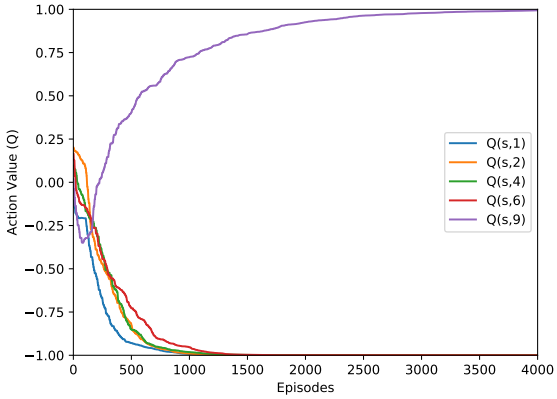


Figure 7: The predicted Q-values during the training phase of tabular Q-learning via self-play.

4.1.3 Use Q-function to Play Tic-Tac-Toe versus a Random agent

In this experiment [implementation] we used tabular Q-learning via self-play to learn the Q-values of all game states of Tic-Tac-Toe, and used them to play the game against a random player. In order to do that we set the starting state of the MDP as the empty board. We evaluated the performance as training progressed by having a π_{greedy} policy (using the Q-function being learned) play 1,000 matches every 1,000 episodes against a player that makes uniformly random moves.

The results are presented in Figure 8. We can observe how after around 70,000 episodes it learned to play the game almost perfectly against a random player. While the random opponent is obviously a weak opponent, is a simple test to verify that we can use the learned Q-values to play the game.

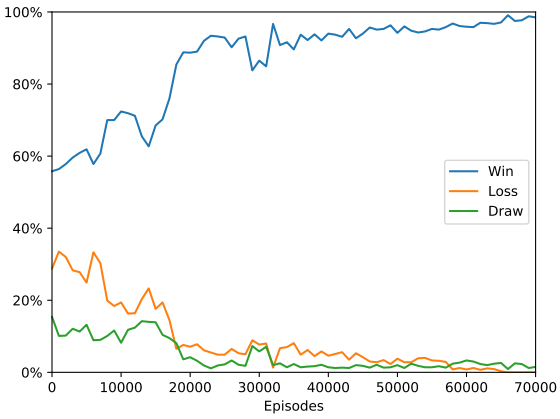


Figure 8: Win/Loss/Draw percentages of playing 1,000 games every 1,000 episodes between a greedy player that uses the learned Q-function against a Random player.

4.1.4 Learning to Play Connect 4 using Q-learning with a Deep Neural Network

Once we validated that our implementation of Q-learning is working and showed that it can learn a tabular value function to play Tic-Tac-Toe we moved into the more complex Connect 4. In this game, storing the Q-values for every board position in a lookup table is not feasible because of its huge state space. Instead of using tabular Q-learning, we used approximate Q-learning [implementation] with a deep neural network (DNN) [implementation]. This approach consists in parameterizing an approximate action-value function, $Q(s, a; \theta_i) \approx Q(s, a)$, in which θ_i are the weights of the neural network.

In some games it can be more efficient to consider the *afterstate value* $V(s \circ a)$, which is the state reached after taking action a in state s, instead of the Q-value $Q(s, a)$ (Sutton and Barto, 1998). We used this approach by using a DNN that takes the state of the board in the input layer, as shown in Figure 9, and outputs the value of being in that state, $V(s)$. After the input layer, we used convolutions with 4 layers of 64 filters of size 3×3 and rectified linear non-linearities. In addition to the convolutions, we added a densely-connected layer with 256 units. Finally, the output layer has only one unit that fires the value of the given board with a *tanh* activation function to keep it in the range $[-1, +1]$.

In this experiment [implementation] our goal was to learn to play Connect 4 starting from 8-ply positions from the UCI Connect 4 dataset (Section 2.1.3). In these positions we already know the game-theoretical outcome of the game assuming perfect play from both players, however, we did not use the outcome for training (no supervised learning), only for evaluation. This helped us to start every episode of Q-learning training from different board positions, which is helpful to learn to generalize by seeing all kinds of board positions.

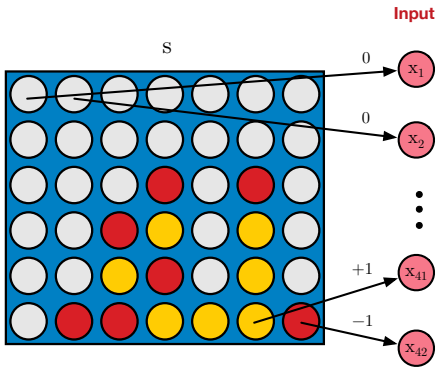


Figure 9: Feature extraction of a Connect 4 board

In Figure 10 we can see how the quality of the target policy improved during training. The policy was evaluated during training every 500 episodes by playing 3,000 games against a random player starting from randomly selected 8-ply positions from the UCI Connect 4 dataset. The first 1,000 games started from positions where the outcome is a win for the first player assuming perfect play from both sides. Figure 10a shows how the policy improved from a 56% to a 94% win rate after 15,000 episodes. This makes sense because the matches started from favorable positions with higher probability of winning for the first player. The next 1,000 games started from game-theoretical draw positions (Figure 10b). As expected, the win rate at the beginning was 50%, but then it went up to 92%. Good improvement, but a little lower win rate than when starting from favorable positions. Finally, the last 1,000 games (Figure 10c) started from unfavorable positions that are guaranteed wins for the second player under optimal play from both sides. In this case the win rate before the policy begins learning is 45%, but then it went up to a 90% win rate. This was also expected, since we started at a disadvantage.

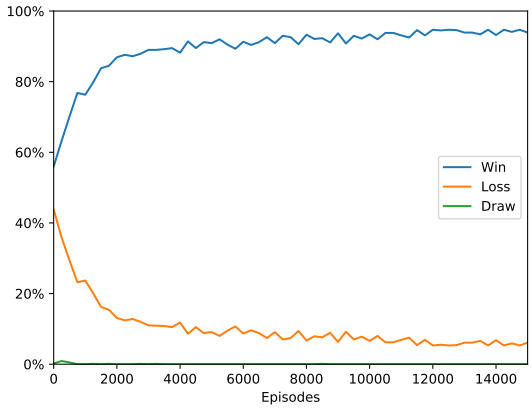
5 Conclusion

5.1 Reflection

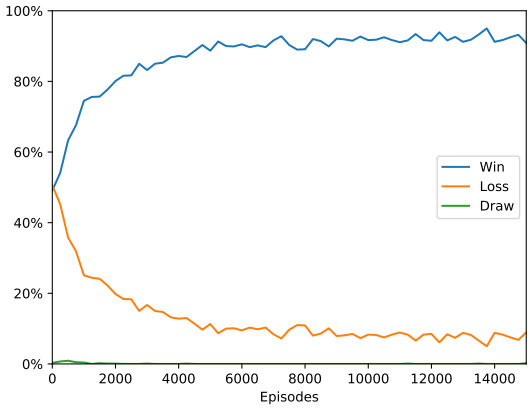
The most difficult aspect of this project was that is extremely hard to stabilize reinforcement learning with non-linear function approximators via self-play. There are plenty of tricks that can be used and hyperparameters that need to be tuned to get it to work, such as:

- **Q-learning**: exploration policy (random?, ϵ-greedy?), discount factor, learning rate, number of episodes.
- **Experience Replay**: batch size, experience pool size.
- **Deep Neural Network**: mlp? convnets? number of layers, number of filters, number of kernels, optimizer (vanilla sgd, adam, rprop), learning rate, pooling, activation functions, etc.
- **ϵ-greedy**: fixed ϵ? decrease ϵ? initial value for ϵ?, how many episodes to decrease it? final value?

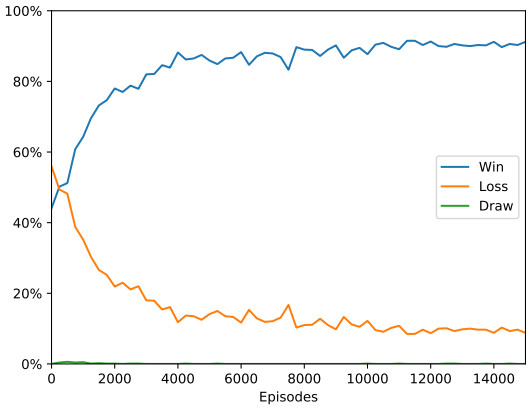
All these techniques and parameters were selected by trial and error, and no systematic grid search was done due to the high computational cost. More than once it seemed that the implementation of the algorithms and techniques was incorrect, and it turned out that the wrong parameters were being used. A "simple" change such as decreasing ϵ, or changing the neural network optimizer made big changes in the performance of the value function.



(a) Starting from game theoretical win positions for player 1



(b) Starting from game theoretical draws



(c) Starting from game theoretical win positions for player 2

Figure 10: Evaluated of the policy during trainingstarting from winning, draws, and loss positions assuming perfect play.

5.2 Improvement

As in most projects, there are way too many aspects that can be improved. But in my opinion the three most important are:

1. **Better benchmarks:** In most of this project we used simple benchmakrks, such as playing against random players. While testing against random is probably the first thing to test against (if you can't beat a random player your learning algorithm is not working), it would be better to find a few heuristics and better players that can be used for testing.

2. **Incorporate other RL techniques:** The field of RL has been advancing fast in recent years. There are a few new and old techniques that I would like to try, such as asynchronous RL, double Q-learning, prioritized experience replay.

References

François Chollet et al. Keras. https://github.com/fchollet/keras, 2015.

R. Colbert et al. Natural language processing (almost) from scratch. In *Journal of Machine Learning Research*, volume 12, pages 2493–2537, 2011.

G. Hinton et al. Deep neural networks for acoustic modeling in speech recognition. In *IEEE Signal Processing Magazine*, volume 29, pages 82–97, 2012.

A. Krizhevsky, I. Sutskever, and G. Hinton. Imagenet classification with deep convolutional neural networks. In *Proc. Advances in Neural Information Processing Systems*, volume 25, pages 1090–1098, 2012.

Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *Nature*, pages 436–444, 2015.

Yann LeCun. *Generalization and network design strategies*. Elsevier, 1989.

M. Lichman. UCI machine learning repository, 2013. URL http://archive.ics.uci.edu/ml.

Long-Ji Lin. *Reinforcement Learning for Robots Using Neural Networks*. PhD thesis, Carnegie Mellon University, 1993.

V. Mni et al. Human-level control through deep reinforcement learning. *Nature*, pages 529–533, 2015.

A. L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal*, 3(3):210–229, 1959.

D. Silver et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–489, 2016.

R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1998.

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

G. Tesauro. Temporal difference learning and td-gammon. *Communications of the ACM*, 38(3), 1995.

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