Machine Learning Nanodegree
Capstone Report

David A. Robles

March 10, 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 to play these games without using any domain knowledge.

1.2 Problem Statement

In this project we use deep reinforcement learning to create an agent that learns to play the game of Connect 4 by playing games against itself. We define the machine learning problem as follows:

- **Task:** Playing Connect 4.
- **Performance:** Winning percentage when playing against other agents, and accuracy of the predicted outcomes on the UCI 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.

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. Alpha-Beta player) using the trained deep neural network as the action-value function to take greedy actions.
- **Prediction accuracy.** The learned action-value function is used to predict the game-theoretic outcomes (win, loss or draw) of the board positions in the Connect 4 Data Set.

2 Analysis

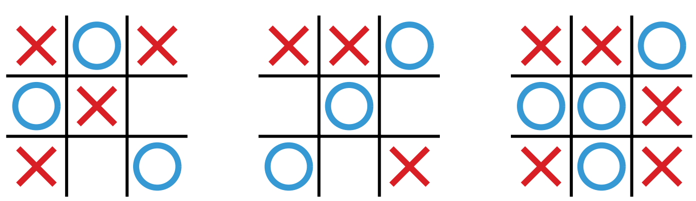
2.1 Data Exploration

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

Implementation

Tic-Tac-Toe is a paper-and-pencil game for two players, X and O, who take turns marking the spaces in a 3x3 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 for X **(b)** Win for O **(c)** Draw

Figure 1: Examples of wins, losses and draws in Tic-Tac-Toe

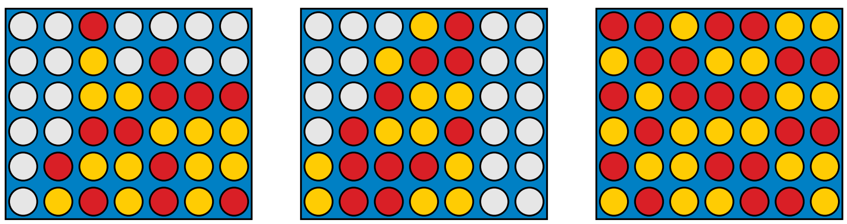
As an example, we will simulate 10,000 games using a random players.

```
from capstone.game import TicTacToe
game = TicTacToe()
game.legal_moves() # [1, 2, 3, 4, 5, 6, 7, 8, 9]
print(game)
```

2.1.2 Connect 4 Environment

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 2 shows three Connect 4 game positions.



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

Figure 2: Examples of wins, losses and draws in Connect Four

2.1.3 UCI Connect 4 Data Set

As part of the testing phase, we will use the *Connect 4 Data Set*[1] that is available from the UCI Machine Learning Repository (Hettich and Merz, 1998). This 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 $\{\text{X}, \text{O}, \text{B}\}$, 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 $\{\text{Win}, \text{Loss}, \text{Draw}\}$. There are 44,473 wins, 16,635 losses and 6,449 draws.

2.2 Exploratory Visualization

2.2.1 UCI Connect 4 Data Set

As an example, here are ten instances selected randomly:

Figure 3 shows a visual representation of the 10 instances of the data set.

2.3 Algorithms and Techniques

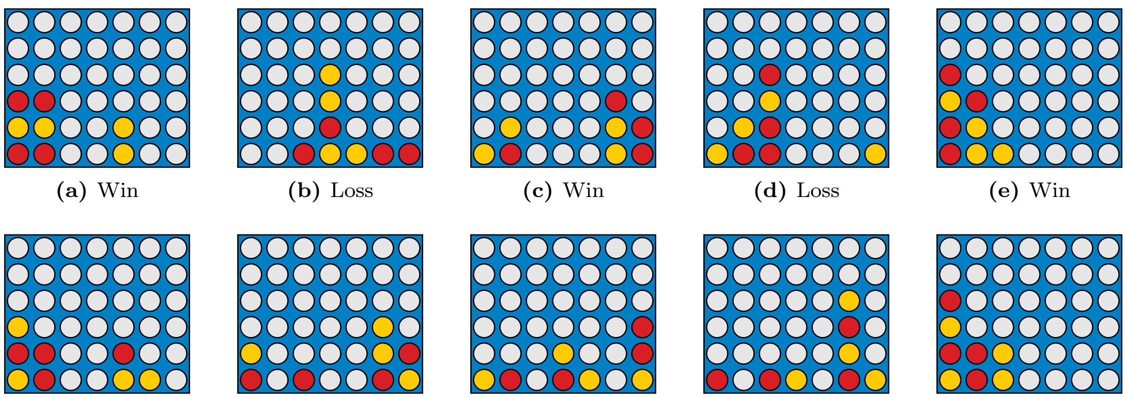
2.3.1 Alpha-beta pruning

Implementation

[1] https://archive.ics.uci.edu/ml/datasets/Connect-4

	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)** Win

(f) Draw **(g)** Win **(h)** Win **(i)** Loss **(j)** Draw

Figure 3: 10 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).

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

Implementation

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:

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

3.3.2 Greedy policy

Implementation

Implementation

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

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

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,

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

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

3.4 Game

Informally speaking, a perfect-information game in extensive form is a tree in the sense of graph theory, in which each node represents the current state of the game, each edge represents a possible action, and the leaves represent the terminal states, which contains final outcomes through a utility function. In the field of AI these are known simply as *game trees*.

A **perfect-information extensive-form game** is defined as a tuple $G = (n, \mathcal{S}, \mathcal{S}_\mathcal{T}, \mathcal{A}, \rho, T, U)$, where:

- n is the *number of players*;
- $\mathcal{S}$ is the set of *states*;
- $\mathcal{S}_T \subset \mathcal{S}$ is the set of *terminal states*;
- $\mathcal{A}$ is the set of *actions*. The set of actions that are available in some particular state $s_t \in \mathcal{S}$ for the player in turn $\rho(s_t)$ 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.
- $\rho : \overline{\mathcal{S}_T} \to \{i \in \mathbb{N} : 1 \leq i \leq n\}$ is the *player function*, which assigns to each non-terminal state $s_t \in \overline{S_T}$ a player i who chooses an action at that state;
- $T : \overline{\mathcal{S}_T} \times \mathcal{A} \to \mathcal{S}$ is the *transition function*, which maps a non-terminal state and an action to a new state;
- $U : S_T \to \mathbb{R}^n$ is the *utility function*, which returns a vector of payoffs $U(s_T) = (u_1, \ldots, u_n)$ for each of the n players of the game. We will use the notation $U_i(s_T)$ to represent the payoff for the ith player when the game is over at state $s_T \in S_T$.

In two-player perfect-information extensive-form games, a *ply* refers to one turn taken by one of the players.

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 GameMDP converter

Converting an MDP

5 Examples

5.1 Estimate state-action values using tabular Q-learning

In this example we will estimate the state-action values for two very simple board positions for both Tic-Tac-Toe and Connect 4. We will be using the tabular version of Q-learning. The goal here is simply to verify that our implementation of the Q-learning algorithm is working as expected. We chose very simple positions because the expected q-values are easy to understand.

We converted both games as MDPs by making the opponent a fixed agent that takes optimal moves by using the Alpha-Beta algorithm. This allowed us to verify that we are learning the q-values assuming the worst case scenario. In other situations in which the depth of the game tree is huge Alpha-Beta would need to combined with a good heuristic evaluation function, but in these simple positions it won't be necessary.

The rewards given by the Tic-Tac-Toe environment are $+1$ for a win for the first player, -1 for a win for the first player, and 0 for a draw.

5.1.1 Tic-Tac-toe

Implementation

For Tic-Tac-Toe we used the board position in Figure 4 as the starting state, s_0. It has five available actions, $\mathcal{A}(s_0) = \{1, 2, 4, 6, 9\}$. By looking at such a simple position we can clearly identify that the best available action is 9, which leads to a win if we keep playing the best possible actions. Any other action (i.e. 1, 2, 4 or 6) would lead to a loss, assuming perfect play from the opponent.

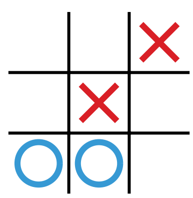


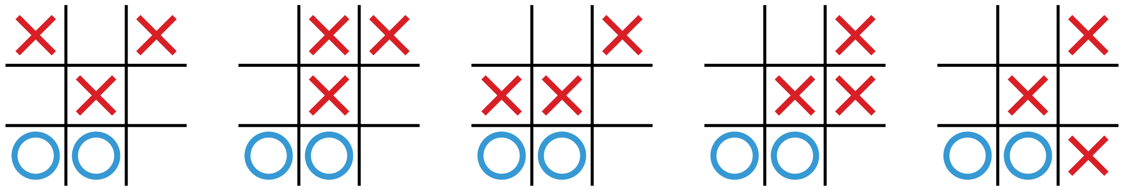
Figure 4: Starting Tic-Tac-Toe state, s_0, with five legal actions, $\mathcal{A}(s_0) = \{1, 2, 4, 6, 9\}$.

We ran the tabular version of the Q-learning algorithm with $\alpha = 0.1$ (learning rate), $\gamma = 1.0$ (discount factor), and a uniformly random behavior policy. The Q-values converged to the expected values after around 800 episodes of training, and are shown in Figure 5. As we can see in Figure 5e, the expected future reward of taking action 9 is correctly estimated as 1.0, since that would lead to a win. At the same time, the q-values for all the other actions was -1.0. This makes sense, since any action not being 9 would lead to an immediate loss. Also, we can see in Figure 6 we can see the predicted q-values for each available action. It is interesting to see how the q-value for $Q(s_0, 9)$ takes some time to converge. This is probably because taking action 9 leads to an eventual win, but not immediately. It requires making another move (either 1 or 6) to win.

5.1.2 Connect 4

Implementation

For Connect 4 we used the board position in Figure 7 as the starting state, s_0. It has three available actions, $\mathcal{A}(s_0) = \{\mathtt{d}, \mathtt{f}, \mathtt{g}\}$. By looking at such a simple position we can clearly identify that the best available



(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

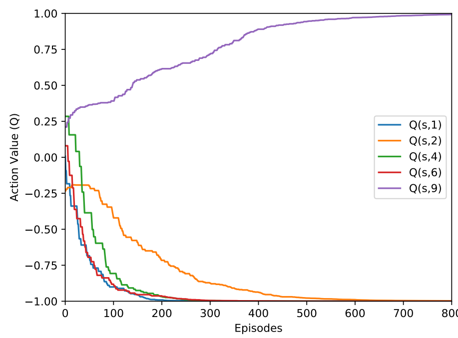


Figure 6: The predicted state-action value during training.

action is `f`, which leads to a win if we keep playing the best possible actions. Any other action (i.e. `d` or `g`) would lead to a loss, assuming perfect play from the opponent.

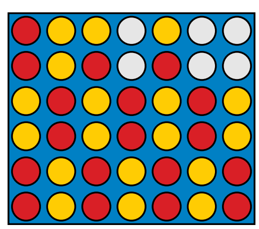
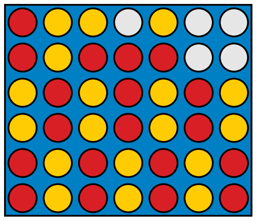


Figure 7: Connect 4 board, s_0, with three legal actions: $\{\mathtt{d}, \mathtt{f}, \mathtt{g}\}$.

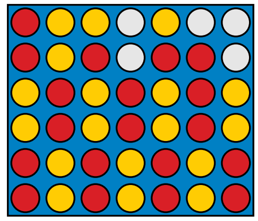
We ran the same tabular version of the Q-learning algorithm with the same parameters as the used for Tic-Tac-Toe $\alpha = 0.1$ (learning rate), $\gamma = 1.0$ (discount factor), and a uniformly random behavior policy. The Q-values also converged to the expected values after around 800 episodes of training, and are shown in Figure 8. As we can see in Figure 8b, the expected future reward of taking action `f` is correctly estimated as 1.0, since that would lead to a win. At the same time, the q-values for all the other actions was -1.0. This makes sense, since any action not being `f` would lead to an immediate loss. Also, we can see in Figure 9 we can see the predicted q-values for each available action.

5.2 Estimate q-values using Q-learning via self-play

In the previous example we estimated the q-values for Tic-Tac-toe and Connect 4 positions against a fixed Alpha-Beta opponent. While this works very well sometimes we don't have that algorithm. An alternative is to use self-play, which consists in making playing games against itself.

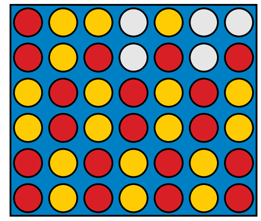


(a) $Q(s_0, \mathtt{d}) = -1.0$



(b) $Q(s_0, \mathtt{f}) = 1.0$

0



(c) $Q(s_0, \mathtt{g}) = -1.0$

Figure 8: All the next states and their q values.

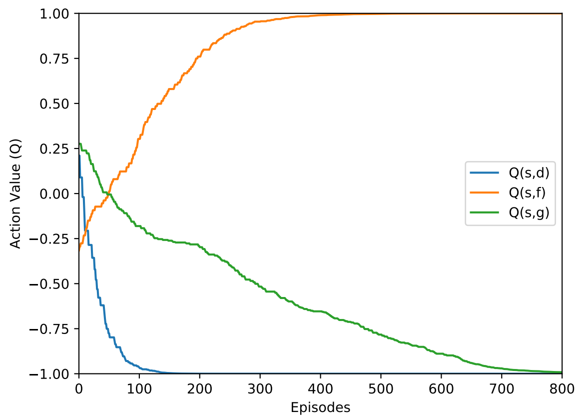
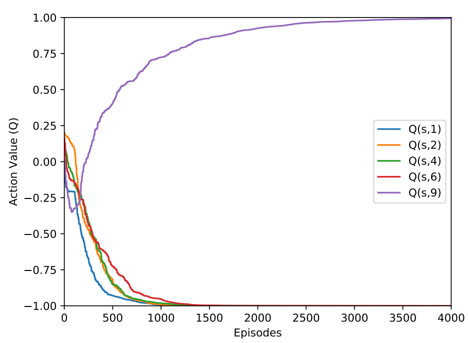
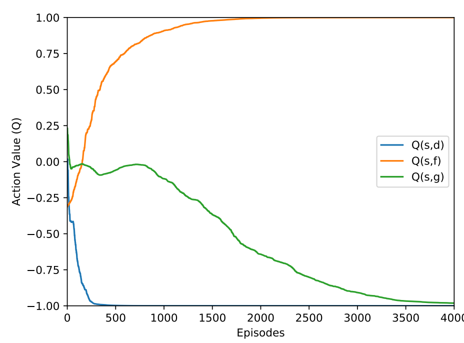


Figure 9: The predicted state-action value during training.



(a) Tic-Tac-Toe



(b) Connect 4

Figure 10: All the next states and their q values.

In both examples we can see that it took longer to learn the episodes because it didn't have a perfect guidance on what was good and bad, it had to learn it by itself. This is a powerful technique that will be used in the next sections to learn a value function for Connect 4. The values are the same as the ones shown in Figures 5 and 8.

5.3 Estimate all q-values using tabular Q-learning

Implementation

Now the idea is to learn to play the games in general starting from the same position. This means that we will use Q-learning to estimate the q-values from the empty board of the game. We will do it for Tic-Tac-Toe. We ran the experiment using similar parameters, and after 65,000 episodes the q-values were found correctly.

In Figure 11 we can see how after 65,000 episodes it learned the values.

We were able to do that because the full game-tree of Tic-Tac-Toe is around 888888 game states. And based on the results we can assume that the majority of the states were visited. But the problem now is that Connect 4 has INSERT game states. That makes it impossible to run the Q-learning algorithm with a tabular Q-function to save the exact value for every single state of the game. The solution to this is now to use a function approximator.

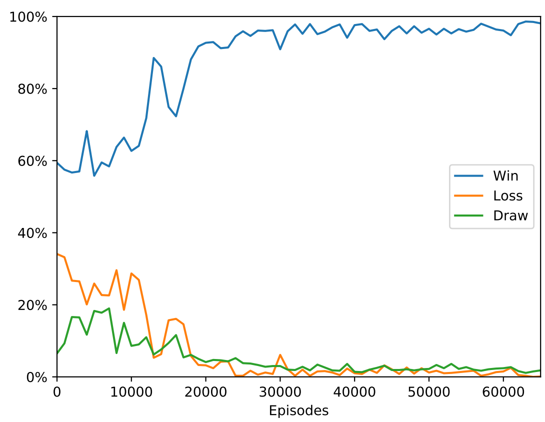


Figure 11: Results after playing 1,000 games every 1,000 episodes against a Random player.

sd fas dfas f sa fsad f asd fa s sd fas dfas f sa fsad f asd fa s sd fas dfas f sa fsad f asd fa s sd fas dfas f sa fsad f asd fa s sd fas dfas f sa fsad f asd fa s sd fas dfas f sa fsad f asd fa s sd fas dfas f sa fsad f asd fa s sd fas dfas f sa fsad f asd fa s sd fas dfas f sa fsad f asd fa s sd fas dfas f sa fsad f asd fa s sd fas dfas f sa fsad f asd fa s sd fas dfas f sa fsad f asd fa s sd fas dfas f sa fsad f asd fa s sd fas dfas f sa fsad f asd fa s sd fas dfas f sa fsad f asd fa s sd fas dfas f sa fsad f asd fa s sd fas dfas f sa fsad f asd fa s.

sd fas dfas f sa fsad f asd fa s sd fas dfas f sa fsad f asd fa s sd fas dfas f sa fsad f asd fa s sd fas dfas f sa fsad f asd fa s sd fas dfas f sa fsad f asd fa s sd fas dfas f sa fsad f asd fa s sd fas dfas f sa fsad f asd fa s sd fas dfas f sa fsad f asd fa s sd fas dfas f sa fsad f asd fa s sd fas dfas f sa fsad f asd fa s sd fas dfas f sa fsad f asd fa s sd fas dfas f sa fsad f asd fa s sd fas dfas f sa fsad f asd fa s sd fas dfas f sa fsad f asd fa s sd fas dfas f sa fsad f asd fa s sd fas dfas f sa fsad f asd fa s sd fas dfas f sa fsad f asd fa s sd fas dfas f sa fsad f asd fa s.

5.4 Estimate all q-values using tabular Q-learning via Self-Play

Implementation

In Figure 12 we can see how after 70,000 episodes it learned the values.

sd fas dfas f sa fsad f asd fa s sd fas dfas f sa fsad f asd fa s sd fas dfas f sa fsad f asd fa s sd fas dfas f sa fsad f asd fa s sd fas dfas f sa fsad f asd fa s sd fas dfas f sa fsad f asd fa s sd fas dfas f sa fsad f asd fa s sd fas dfas f sa fsad f asd fa s sd fas dfas f sa fsad f asd fa s sd fas dfas f sa fsad f

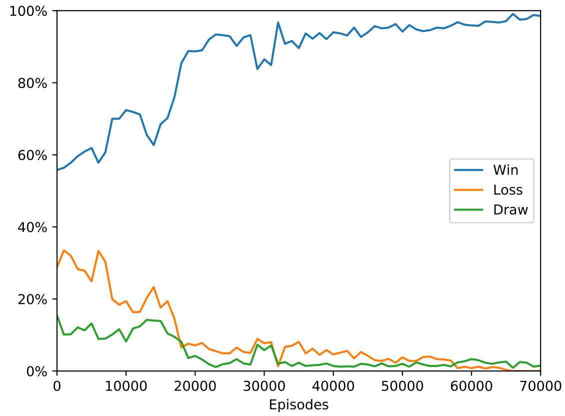


Figure 12: Results after playing 1,000 games every 1,000 episodes against a Random player.

asd fa s sd fas dfas f sa fsad f asd fa s sd fas dfas f sa fsad f asd fa s sd fas dfas f sa fsad f asd fa s sd fas dfas f sa fsad f asd fa s sd fas dfas f sa fsad f asd fa s sd fas dfas f sa fsad f asd fa s sd fas dfas f sa fsad f asd fa s.

sd fas dfas f sa fsad f asd fa s sd fas dfas f sa fsad f asd fa s sd fas dfas f sa fsad f asd fa s sd fas dfas f sa fsad f asd fa s sd fas dfas f sa fsad f asd fa s sd fas dfas f sa fsad f asd fa s sd fas dfas f sa fsad f asd fa s sd fas dfas f sa fsad f asd fa s sd fas dfas f sa fsad f asd fa s sd fas dfas f sa fsad f asd fa s sd fas dfas f sa fsad f asd fa s sd fas dfas f sa fsad f asd fa s sd fas dfas f sa fsad f asd fa s sd fas dfas f sa fsad f asd fa s sd fas dfas f sa fsad f asd fa s sd fas dfas f sa fsad f asd fa s sd fas dfas f sa fsad f asd fa s.

5.5 Estimate Q-values Using Approximate Q-learning

sd fas dfas f sa fsad f asd fa s sd fas dfas f sa fsad f asd fa s sd fas dfas f sa fsad f asd fa s sd fas dfas f sa fsad f asd fa s sd fas dfas f sa fsad f asd fa s sd fas dfas f sa fsad f asd fa s sd fas dfas f sa fsad f asd fa s sd fas dfas f sa fsad f asd fa s sd fas dfas f sa fsad f asd fa s sd fas dfas f sa fsad f asd fa s sd fas dfas f sa fsad f asd fa s sd fas dfas f sa fsad f asd fa s sd fas dfas f sa fsad f asd fa s sd fas dfas f sa fsad f asd fa s sd fas dfas f sa fsad f asd fa s sd fas dfas f sa fsad f asd fa s sd fas dfas f sa fsad f asd fa s sd fas dfas f sa fsad f asd fa s sd fas dfas f sa fsad f asd fa s sd fas dfas f sa fsad f asd fa s.

sd fas dfas f sa fsad f asd fa s sd fas dfas f sa fsad f asd fa s sd fas dfas f sa fsad f asd fa s sd fas dfas f sa fsad f asd fa s sd fas dfas f sa fsad f asd fa s sd fas dfas f sa fsad f asd fa s sd fas dfas f sa fsad f asd fa s sd fas dfas f sa fsad f asd fa s sd fas dfas f sa fsad f asd fa s sd fas dfas f sa fsad f asd fa s sd fas dfas f sa fsad f asd fa s sd fas dfas f sa fsad f asd fa s sd fas dfas f sa fsad f asd fa s sd fas dfas f sa fsad f asd fa s sd fas dfas f sa fsad f asd fa s sd fas dfas f sa fsad f asd fa s sd fas dfas f sa fsad f asd fa s sd fas dfas f sa fsad f asd fa s sd fas dfas f sa fsad f asd fa s sd fas dfas f sa fsad f asd fa s.

6 Results

6.1 Model Evaluation and Validation

Content.

6.2 Justification

Content.

7 Conclusion

7.1 Free-Form Visualization

Content.

7.2 Reflection

Content.

7.3 Improvement

Content.

References

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

C. B. S. Hettich and C. Merz. UCI repository of machine learning databases, 1998.

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

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

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.

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.

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.