Building an AI to play Tic-Tac-Toe with a Q-learner (Markov Decision Process)

By: Aron Tharp

Abstract:
A very simple exploration in training an AI by using a Q-learner (Markov Decision Process), starting with a purely random AI and improving iteratively.

Summary:
As a proof of concept, I developed an AI to play the classic game of tic-tac-toe. To train the AI, I begin with a purely random AI which chooses at random where to play an x or o. At the end of each match, the winning player receives a positive reward, the losing player receives a (large) negative reward, or both players share a very small reward if the result is a tie. The board states of each match are recorded along with their rewards, with the final move receiving the full reward and each preceding move receiving progressively less of the reward.

Q-learner:

Simulation

By definition, a Q-learner is 'model-free'. This means that no logic was used when determining a move except prior results or pure randomness. To train the learner, 2.5 million games were played. The amount of randomness in the play style of each player was progressively decreased over time per the pseudo code in figure 1 below. Note that the max function is ensuring that there is at least a 10% chance of playing randomly during training.

```
randomness = max(1 – (games played / target games played), .1)
if (random number between  0 and 1 < randomness) :
   play a random move
else:
   choose the move which is known to generate the highest reward
```

figure 1 – pseudo code to determine when to play randomly

Rewarding the AI

When a player wins the match, they receive a reward of 10 points. These are then attributed to the resulting board after each move that player made during the match. The final move receives the full amount, and then each prior move receives a discounted reward with a discount rate of 4%. The same discounting applies when the player loses or ties, with -20 and 1 being the respective rewards of those outcomes. When the reward for a board state is updated (after each match), the new reward is given a weight of 20% and the prior rewards are given a weight of 80%. Over time, a steady state is reached where the updated values are not substantially different from the prior values and simply move within a small range. A basic formula for the update is shown below in figure 2.

```
for the board states after X played:
   expected reward of last board state = 80% * prior rewards + 20% * new reward * (1-.04*0)
   expected reward of 2nd-to-last board state = 80% * prior rewards + 20% * new reward * (1-.04*1)
   expected reward of 3rd-to-last board state = 80% * prior rewards + 20% * new reward * (1-.04*2)
   etc.
for the board states after O played:
   (same as for X)
```

figure 2 – pseudo code to update rewards for each board state

Illustration
Let's hypothetically step into the training during the one millionth match out of 2.5 million. At each turn, the player will have a 1-(1million/2.5million) chance of playing randomly, or 60% chance. If the turn happens to fall into the 40% chance to play based on experience, the program will consider each possible move. Let's assume we are at the board state in figure 3 below.

X O
O
X X

figure 3 – sample board state

It is O's turn. There are 4 empty spaces, so 4 potential boards will be considered. Whichever board is known to generate the highest reward will be chosen. Since losing generates the reward of the largest magnitude (-20), then if the prior experience has shown that a specific move is likely to generate a loss then it will not be made. In figure 2, we see that X can win by playing in the bottom left corner. If the new boards have been played, and the other 3 have shown to generate a loss by X playing the bottom left corner, then our learner will know that playing in the bottom left corner is the best play. However, if the other new boards have not yet been played, or if they have been played but X did not play in the bottom left corner, then our learner may not yet know that the best move is to block X.

References:

https://en.wikipedia.org/wiki/Q-learning
https://en.wikipedia.org/wiki/Markov_chain
https://en.wikipedia.org/wiki/Tic-tac-toe

Contents:

Tic-Tac-Toe_Play-the-Game.ipynb
- Play the game against the 2 AIs (AI 1 is the random player and AI 2 is the fully trained AI)
- 2 human players are also an option
- Play vs. the trained AI requires the x_plays and o_plays directories with their contents

Tic-Tac-Toe_QLearning_Trainer.ipynb
- Simulates matches between AIs using the Q-learning logic
- Outputs x_plays.txt and o_plays.txt used in Tic-Tac-Toe_Play-the-Game.ipynb