Game Domination: Q-Learning for Tic-Tac-Toe and Checkers

Peter Hickman, Stephen Albro, Vincent Chow, & Brandon Price

Demo Video | Github Repository

Overview

We implemented Q-Learning for tic-tac-toe and extended our framework to checkers. The algorithm plays itself in order to learn what moves are more likely to result in a winning game state. We also implemented graphical user interfaces to enable a human user to play the algorithm in both games.

See the README file for instructions on how to run the learning algorithm and play tic-tac-toe and checkers.

Planning

In our initial draft specification, we planned to use reinforcement learning for checkers. One week later, in our final draft, we set our sights on tic-tac-toe instead, and considered checkers a “reach” goal. At the start of our project, we decided on the data structures, modules, and functions necessary for our implementation, and figured out how to divide up the work in our implementation. Our initial plans for implementing Q-learning provided a solid foundation for our eventual implementation, though we needed to work on the concrete details.

The work was divided as follows: Brandon worked on graphical user interface and functions associated with interfacing the GUI with the algorithm. Stephen, Vincent, and Peter all worked with the other modules; Peter specialized in the theory of Q-learning; Stephen worked with the Pickle library, and Vincent focused on working with the Q-table.

We achieved most of the milestones from our final draft, the only exception being that we didn’t create alternate versions of the algorithm to compete against each other. We did not successfully implement a particularly skilled checkers-playing algorithm, which we aimed to do in the first draft, but we took first steps to teaching checkers to the computer.

Design and Implementation

Language and Libraries. We used the programming language Python because of its extensive documentation and libraries. For the graphical user interface, we used the built-in Python library Tkinter. For saving our Q-Table data-structure we used the Pickle module.

Q-Learning Implementation. Q-learning is a reinforcement learning algorithm that associates Q-values with states and updates the Q-values throughout the game. In our implementation, in order to learn, the algorithm plays against itself, *exploiting* moves with probability 0.5 and *exploring* moves with probability 0.5. Exploiting means choosing the best move that the algorithm has learned thus far--the move with the best Q-value. Exploring means choosing one of the least-visited moves. After a move has been chosen, the the next state is identified and the the immediate reward associated with this next state is computed.

Next, the associated Q-value for the previous state is updated based on this reward from the next state. After many iterations, the Q-table (storing Q-values) is saved, and the algorithm playing moves according to this table may be a surprisingly formidable opponent.

Q-Learning Lessons from Tic-Tac-Toe and Checkers

Advantages of two games. Implementing Q-learning for two different games taught us how to abstract the fundamentals of Q-learning away from the details and data structures of any given game. For example, in tic-tac-toe, our Q-table easily stores all possible board configurations without significantly impacting run-time. However, the number of possible checker board configurations is *dramatically* higher and would impact runtime and memory; thus, for checkers, we represented each state not as a complete board but as a dict that holds each player's number of men (ordinary checkers) and each player's number of kings.

Problems with Checkers. The tradeoff for better run-time was losing information about the board configuration, which severely limited our ability to learn. In our checkers implementation, the algorithm was essentially *blind*, not differentiating among moves that led to the same number of pieces on the board. As of this writing, the computer is a poor checkers player, but is better than random. It often will jump the opponent's pieces, but sometimes misses advantageous jumps. We hypothesize that this problem could come from (1) too few learning games played (we've only done 10,000) or (2) too-limited representation of states.

Breakdown of substitution model. We encountered a surprising and frustrating problem when manipulating data structures used to store states. After much work seeking uncover the source of mysterious error messages in tic-tac-toe, we discovered that creating a new state changed the *original* state because the new state pointed to the original location in memory. To solve this, we used the Python function "deepcopy" to create a pointer to a new location in memory. Later, while working on our checkers implementation, we encountered a similar problem, and, recognizing the pattern, we used deepcopy again for a fix.

Reflections

Language and Libraries. Our decision to use Python worked well because two of our team-members already had some experience in Python, and also it is an easy language to learn, so the other members were able to pick it up quickly The Tkinter library was very simple and so it worked well for simple games like Tic-Tac-Toe and Checkers. However Tkinter was almost *too* simple at times, such that it made it very hard to allow for pausing before the computer made a turn. Still, overall, Tkinter was definitely a good decision.

Learning Algorithm. We are very proud of our tic-tac-toe results. It is beautiful to see the powerful effects of reinforcement learning. Our group enjoys being unable to defeat our tic-tac-toe monster. The algorithm has learned to force a trap and win as early as move three if the opponent makes the wrong response to its first move. We are also proud of checkers and its graphical interface, though we realize that with more time checkers could really be improved.

Debugging. The most useful debugging tool we utilized in python was "print" statements. Since we implemented Q-learning for board games, we were able to visualize crucial information such as board configurations and actions by simply printing them in the terminal.

Graphical User Interface. If we had more time, we would have given our program a nicer layout. The layout is pretty much only bare-bones right now, so it would be nice to get more creative with it. One issue with our GUI is that it doesn't allow double jumps. This is because it became too complicated to do once we'd started implementing other functions. This taught us that it would have been smart to decide early on not only which functions we were going to implement but also the general pseudocode associated with each function. It is very easy for a GUI program to get very un-organized when you are implementing it from scratch.

What We'd Do Differently. We would have tried to implement Connect 4 rather than checkers. Checkers is too hard. If we'd continued with checkers, we'd have explored different ways to represent a state so that we could have a small Q-table but actually learn.

The Most Important Thing We Learned. We learned that one of the best ways to implement a large project is to carefully devise the general framework early on, then fill in with pseudocode before starting the critical work of the actual implementation. Through this process, each team member can focus on the implementation of a specific section, and the project has unity and direction from the beginning.

Advice for Future Students

Stay organized from the beginning. First, begin by setting up your git repository and make sure everyone can pull/push to it. Then decide on the programming language and all necessary libraries. Decide what type and what modules need to be implemented and what functions go inside these modules. Then facilitate who should implement which modules. If you stay organized from the beginning, your project will go fairly smoothly and you'll be able to handle whatever bumps come along the way. We had a fairly smooth and successful project experience because of our organization.

Consistency and Communication. Make sure there is excellent communication between whoever is implementing GUI and whoever is implementing the back-end algorithms. The GUI designer may find that his functions are consistent and work fine, while the algorithm designers find their inputs and outputs problematic or at least inefficient. This was only a minor issue for us, but there is room for improvement and future students should know this. We have come to appreciate the value of consistency, factoring out code, and keeping it simple.

Spiral Development. Implement something feasible first, and then extend to a more complex implementation. We found implementing tic-tac-toe first a good strategy, teaching us the fundamentals of Q-learning and providing a framework easily extendable to checkers.

Pick a Fun Topic. Watching our program learn was an absolute joy. We jumped up and down, high-fived each other, and laughed when the computer tic-tac-toe player finally started blocking its opponent. These joys made most of our hours of work very pleasant. :)