Solving JumpIN' Using Zero-Dependency Reinforcement Learning (Student Abstract)

Rachel Ostic,[1] Oliver Benning,[1] Patrick Boily[2]
[1, 2]University of Ottawa
[2]Data Action Lab, Ottawa
[2]Idlewyld Analytics and Consulting Services, Wakefield
{rosti049, obenn009, pboily}@uottawa.ca

Abstract

Reinforcement learning seeks to teach agents to solve problems using numerical rewards as feedback. This makes it possible to incentivize actions that maximize returns despite having no initial strategy or knowledge of their environment. We implement a zero-external-dependency Q-learning algorithm using Python to optimally solve the single-player game JumpIn' from SmartGames. We focus on interpretability of the model using Q-table parsing, and transferability to other games through a modular code structure. We observe rapid performance gains using our backtracking update algorithm.

Introduction

Reinforcement learning (RL) is a machine learning technique that does not require complex models or close supervision, only numerical feedback. This makes it highly applicable to solving puzzle games, where it is not computationally feasible to model strategies using greedy look-ahead techniques, and a winning strategy may not be known.

In an RL setting, an agent typically finds itself in a state with a set of possible actions. The agent has contextual information from its environment; this knowledge may be complete or partial. Based on the agent's actions and their subsequent outcome, it is given feedback, known as a "reward", to encourage or discourage similar behavior in the future. The agent's goal is to maximize its overall reward as it selects actions, thereby growing more competent at its task.

We test an RL approach on JumpIN', a single-player puzzle game.[1] It is played on a 5×5 board with three piece types: the bunny, the fox, and the mushroom. Mushrooms are stationary during gameplay. Foxes can move forward and back in their row if they are unimpeded. Bunnies can move in all four directions, but only if they are jumping over other pieces. The game ends when all the bunnies on the board are in burrows, which can be found in the center and at each corner of the board. The game manual includes 60 sample puzzles ranging in level from very simple ($\sim$ 5 moves to win) to very challenging ($\sim$ 100 moves to win). The appeal of this game is that the rules are easily explainable, yet the puzzles themselves are not easy. We want to know if a Q-learning-trained agent can yield a human-interpretable strategy. Furthermore, it is easy to scale the game to larger board sizes or increase the number of movable pieces, an interesting avenue for analyzing how RL solutions scale on more complex games.

Methods

While there exist frameworks in which to perform RL, not all of these make models reusable from one session to another or make it possible to examine them after training to distill strategic insight. Taking a simple game with the potential for generalizations gives an ideal opportunity to test and compare solution strategies. We leverage the object-oriented nature of Python to create an independent, flexible and easy-to-understand encoding of the JumpIN' game.

Code Structure The modular code naturally separates the main tasks of game play. We designed three modules: game mechanics, solution, and training. The game mechanics module handles game board initialization, and allows the agent to query available actions. Once one has been selected, the agent calls on this module again to perform the move. Keeping this part independent means that if we wanted to change rules, expand on the game, or even encode a different game, it would be straightforward.

The solution module takes an initial board configuration and grows a solution tree until a winning state is found. Previously encountered states are pruned to ensure that this algorithm will terminate. To determine which nodes to add, we use a library of sorters, i.e. possible strategies to help the agent select actions. These allow us to test and compare different strategies both with and without RL training to gain a better understanding of what an optimal approach to JumpIN' looks like.

The training module facilitates the creation of template models, importing and exporting of models from files in the JSON format, and performing Q-learning training on them. Repetitive calling of the solution module with the appropriate sorters and callbacks is encapsulated into train and test methods, making it simple to train on a selected puzzle, tune the Q-learning parameters, or test a model's performance.

Q-learning Implementation Q-learning views the problem as a sequence of states, actions, and rewards. At time t,

[1] SmartGames. 2020 (accessed Sept. 16, 2020). JumpIN'. https://www.smartgames.eu/uk/one-player-games/jumpin

the agent is in state S_t, chooses action a_t and receives reward r_t. As a result, it transitions to state S_{t+1} and must select a new action denoted a_{t+1}. The agent continues until it reaches a JumpIN' win state. In state S_t, the agent may choose among multiple actions. To rank them, we define Q-values $q(S, a)$ for each state-action pair. At the start of training, these are all uniformly initialized to zero. For each action taken by the agent, we apply the Bellman equation:

$$\begin{aligned} q_{\text{updated}}(S_t, a_t) &= q(S_t, a_t) \\ &+ \alpha \left(r_t + \gamma \max_{a \in \{a_{t+1}\}} q(S_{t+1}, a) - q(S_t, a_t) \right) \end{aligned}$$

where α, the learning rate, and γ, the discount factor, are tunable parameters to quantify how much the agent takes into account new information, and whether to seek immediate or long-term rewards. Actions are selected with ϵ-greedy policy during training. We also include the option to set $\epsilon = 0$ to "test" the model's default solution path.

Our Q-learning approach prioritizes efficient solutions: since the solution module can always find a solution, training should more highly recompense those requiring fewer moves. We attribute a single reward upon winning; its value is the reciprocal of the number of moves in the solution. We also perform the Q-value updates in a backtracking manner, following the states in the solution tree from win back to start. The advantages of this technique are that the final reward filters up through the encountered states after just one training episode via the $\max q(S_{t+1}, a)$ term, and that training can be implemented as a post-win callback without modifying the solution module.

Results and Discussion

We verify that the test solutions converge to near-optimality within as few as 20 training iterations. Parameters α, γ and ϵ are varied to observe their effects on training. Figure 1 shows that $\epsilon = 0.99$ rapidly yields an efficient solution, but the Q-table is exploited so rarely that random moves delay the minimal path being strongly reinforced, leading to oscillations. With $\epsilon = 0.1$ the solution length decreases slowly and monotonically. We observe that changing α has little effect on results, and setting $\gamma = 0.99$ leads to divergent behavior.

To generate a large sample Q-table, we trained on each of the 60 sample puzzles 10^4 times, filling the table with 92,183 states and an average of 4.7 possible actions per state. Of these actions, 55% still have a Q-value initialized to zero. After training, the test solutions are all within 5 moves of the minimum length. We analyzed for trends by comparing move rankings from the Q-table to intuitive rankings based on board state (e.g. average bunny distance to burrow) or type of move (fox or bunny).

From this analysis, we determine that of those used in the comparison, the strategy best matching the optimal solutions is simply to do moves leading to the greatest possible number of bunnies in burrows. This discrete strategy outperformed the similar sorter based on average bunny distance to burrow. As for fox moves, we notice that they are frequently on the lower end of Q-table rankings, leading us to believe that foxes can stay put until they are needed for bunnies to leap-frog over. In addition, the fraction of foxes out of movable pieces appears to be a good predictor of the fraction of fox moves required in solutions as shown in figure 2.

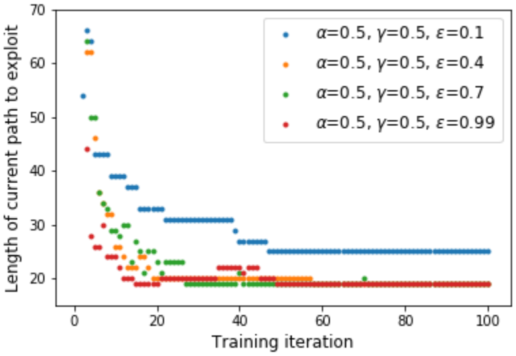

Figure 1: Convergence of Q-table's default solution for different values of exploration rate ϵ used in training. The minimum number of moves is 19.

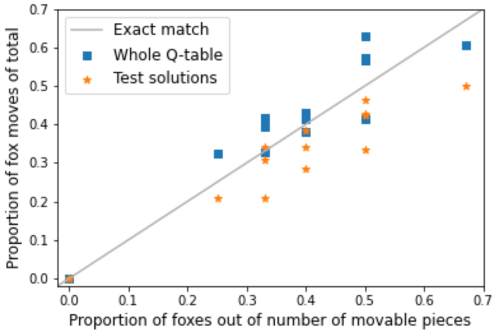

Figure 2: Comparison showing how the fraction of states where a fox move is ranked highest varies with the fox fraction of movable pieces. Squares are averaged over the whole Q-table; stars, over states in test solutions. "Exact match" line is a guide to the eye, not a fit.

Conclusion

We have successfully created a zero-dependency Q-learning codebase that allows us to model a game and interface with it through solution and training modules. After training, we extract some intuitive strategies and verify convergence over a range of parameters. To continue this work, we intend to generalize JumpIN' to larger dimensions with more pieces, and compare this interpretable implementation with deep Q-learning using a neural network to estimate Q-values. This may make enable us to better take advantage of symmetries on the game board.

References

Stephens, D. 2019. Applying Deep Reinforcement Learning to Finite State Single Player Games. CS229 projects, Fall 2019, Stanford University.

Sutton, R. S.; and Barto, A. G. 2020. *Reinforcement Learning: An Introduction*. MIT Press, 2nd edition.