Reinforcement learning on a game of Tic Tac Toe

02901 Advanced Topics in Machine Learning - Project Report

Simon Aertssen (s181603) 13/09/2020

This report aims to discuss the project work for the summer school on advanced topics in machine learning. Different strategies for playing a game of Tic Tac Toe are discussed and compared to each other, in function of the highest number of wins.

1 Introduction

Our aim is to teach an algorithm, the *agent*, to play and win a game of Tic Tac Toe. A game consists of a sequence of discrete states x_k at time k, taken from all available board positions $\mathcal{S}_k = (i, j)\ \forall i, j \in [0, 1, 2]$. The agent has to learn to choose the action u_k from the set of available actions $\mathcal{A}_k(x_k)$ that maximises the chances of winning the game. We can formulate this as a regression of the action *policy* mapping at time k

$$u_k = \mu_k(x_k) \tag{1}$$

The policy π is thus a set of functions:

$$\pi = \{\mu_0, \dots, \mu_{N-1}\}, \mu_k : \mathcal{S}_k \mapsto U_k \tag{2}$$

In a game like Tic Tac Toe, the agent has an *opponent* with the same goal: to win the game! We can thus introduce a noise term w_k that symbolises the action taken by the opponent in the previous turn: $w_k \sim P_k(W_k \mid x_k, u_k)$. These actions can be random (as sampled from P_k), but in most cases P_k is not so much a probability distribution but a policy itself, and W_k is the opponents action space that is left after the agent's turn. We can thus also say that $w_k = \eta(x_k)$. This notation assumes that the agent and the opponent both choose an action at timestep k.

An optimal policy π^* can be learned by minimizing

$$J_\pi(x_0) = \mathbb{E}\left[g_N(x_N) + \sum_{k=0}^{N-1} g_k(x_k, \mu_k(x_k), w_k)\right]$$

which is the expected cost of having played a game by taking actions according to the policy, starting from an empty board x_0. The *control problem* [1] of the game is then stated as:

$$\pi^* = \underset{\pi=\{\mu_0,\dots,\mu_{N-1}\}}{\arg\min} J_\pi(x_0) \tag{3}$$

The code repository can be found here.

2 Strategies

2.1 Random play

The agent draws discrete samples from the action space with equal probability:

$$u_k = \mathcal{U}(\mathcal{A}_k \mid x_k, w_k)$$

This is not a good game strategy, but it is useful to test an agent against this opponent.

2.2 Minimax

The agent will envision all possible future actions and choose the action with the highest reward. This is accomplished by recursively choosing a move from $\mathcal{A}_k$ and imagining what the opponent would do. If a game is envisioned to be won by the agent, the first action of the set of actions leading to the victory will be of highest value among other actions, and this action will be taken. The agent chooses the best action and therefore minimises the cost function (maximises the value of the action); but while when envisioning to be the opponent, it tries to maximise the cost function, as that is the best option from *its* perspective:

$$J_N(x_N) = g_N(x_N)$$

$$J_k(x_k) = \min_{u_k \in U(x_k)} \left[\max_{w_k \in W_k(x_k, u_k)} \left[g_k(x_k, u_k, w_k) + J_{k+1}(f_k(x_k, u_k, w_k))\right]\right]$$

Even for a game of Tic Tac Toe the minimax algorithm yields and exponential number of nodes in the search tree. It also yields a high overhead, as many times the same position might be evaluated, due to exploring *depth first*, though the agent will always be able to at least secure a draw. One way to tackle this problem is to limit the search depth of the algorithm, though an optimal solution is not guaranteed.

2.3 $\alpha\beta$ pruning

A special technique to drastically improve the performance of the minimax algorithm is to keep track of the value of each envisioned game throughout the search tree. We define α as the highest value the agent has awarded to a possible action when following the optimal policy, and β as the lowest value an envisioned opponent is guaranteed. When the value associated with an action is larger than β in the case of the agent, or smaller than α in the case of the opponent, the branch can be pruned. To improve the speed of the algorithm even further, we can save the board state and link it to the move chosen by the algorithm for faster retrieval in future games.

2.4 Tabular learning: state-value pairs

The agent will remember all board states it has seen before and associate a value with reaching that board state by choosing the appropriate action. Choosing a move is as simple as going through all legal actions and choosing the on with the highest value. All values are initialised with a value of 0. This means that the algorithm will always be pessimistic about new moves, reinforcing the value of actions taken in the past. To ensure that new

actions are explored, a certain percentage of moves is a random choice.

Learning the value V of an appropriate action is implemented by means of an incremental update rule:

$$V_{n+1} = V_n + \alpha \left[R_n - V_n \right] \tag{4}$$

where R is the reward of having chosen the action. In this formulation we can recognise a weighed moving average of all rewards that have been awarded in the past [2]. In this context α is the learning rate. A decay of the learning rate can easily be implemented.

The rewards can be chosen to reflect the best desired outcome. Of course winning the game should yield a high reward, but when playing against an optimal player like the minimax algorithm, the best result that can be expected is a draw. We can then also distinguish between rewarding a player for setting up the last moves so the opponent has to choose a draw, and actually choosing the draw. Losing the game can be rewarded with -1 in case we expect the player to win, or 0 in case we do not.

2.5 A neural network

When thinking about the state-value pair, it is clear that the value mapped to a state comes from a highly nonlinear value function, and that we are performing a rough kind of regression in 2.4. In this section we will teach a feed forward neural network to associate a value to each available position on the board and choose the action with the highest value.

The learning process involves two identical networks: a *target* network from which we sample the value function, and a *policy* network on which we perform backpropagation. This is necessary because we sample the target network twice for every board state: once when playing the game to obtain the next move, and once when backpropagating through all the moves that were chosen in the end that led up to the end of the game. After the backpropagation the target net loads the state dictionary of the policy net.

The networks are simple feed-forward networks with one hidden layer and a relu activation function, and a sigmoid function on the output to map the values in a range of {0, 1}. Backpropagation is performed by going through the set of played moves in reverse and interchanging the value of the chosen output with the maximum value on the board at that time, and by setting the value of all illegal moves in that position to zero. For the last move that ended the game, the maximum board value is replaced with the reward associated to the end state of the game. This value map is fed to the loss function for every position and the network weights are updated accordingly.

The implementation is inspired by [3], though I did not use a convolutional network, and the *experience replay* was replaced by playing more games from scratch.

2.6 On human cognition

The minimax and $\alpha\beta$-pruning algorithms are ways of brute-forcing a solution the game, and we cannot expect a human to make decisions fully based on this process. A human would never systematically envision a future game. Instead, we have an inherent ability to think about a strategy and validate (parts of) that through envisioning future moves, both our own actions and the opponents actions.

The tabular method is not a way we think about playing games either. The strategy consists again of remembering the positions encountered in the past and choosing the one with the highest value. It is true that we can see (4) as learning, though there is no connection between the different states. The map we are learning is closer to the map of the minimax algorithm than to a map a human would try to learn.

Finally, what the neural network does is the closest behaviour to that of a human. The board is fed to the agent without any extra information and the agent chooses the appropriate action. Whether the agent really learns a policy is up to debate, but clearly the same value function is evaluated at each turn.

2.7 Game symmetry

The game board of Tic Tac Toe is characterised by many different symmetries, and we can use these symmetries to infer information during the learning process. Most recognisable are the three rotational symmetries r^n of $n\cdot$ 90 degrees (as a rotation of 360 or 0 degrees is an identity mapping), two reflections t_x and t_y about the x and y axes and two reflections t_{AB} and t_{CD} about the main and the off-diagonal. These symmetries are known as the D_4 group.

One last symmetry is to be found in the players, where either playing X or O is arbitrary, as it is only important which player begins. The symmetries are shown in Figure 1.

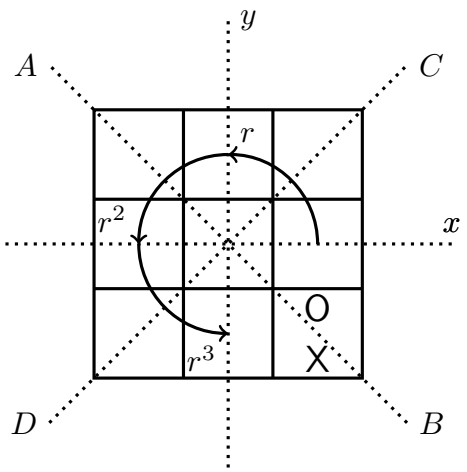

Figure 1: The eight symmetries of the game.

This means that for any state x_k there exist up to eight games with a symmetric board position. Using this information we can reduce the number of states we need to

remember, and thus reduce the workload on the agent, which can now generalize better across the action space $\mathcal{A}_k(x_k)$. However, implementing these symmetries into the different strategies yields a tradeoff: computing the eight symmetric states outweighs the speed at which we can store the low number of games in memory. Therefore this has not been implemented for this project.

3 Results

We will report on the learning process of algorithms where learning the value function is involved. The tabular state-value method is implemented in the `QPlayer` class, and the neural network is implemented in the `SmartPlayer` class. A notebook with the results is available here.

When playing against a random player, both learners perform reasonably well. This is a good check that there is no bias of being generally a bad player.

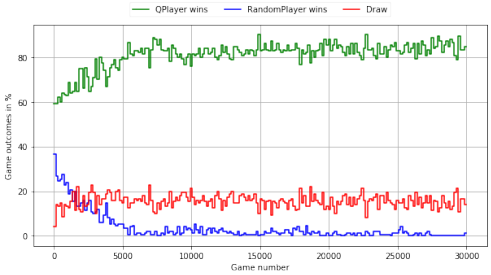

Figure 2: The tabular method quickly learns to consistently defeat the random player. We disregard the last defeats as there is still a chance that the player chooses a random action. This is avoided when testing the network.

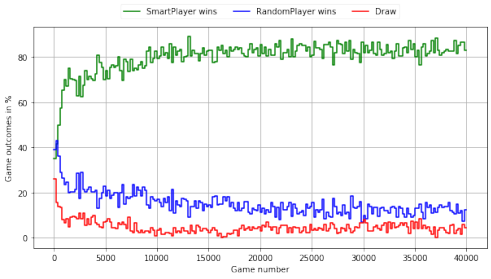

Figure 3: The neural network quickly learns the basics of the game but needs a lot more time to converge.

When playing against the minimax algorithm (or the $\alpha\beta$-pruning heuristic), the `QPlayer` consistently achieves a draw after training. The convergence happens much faster, as there is more to learn from an opponent that always wins.

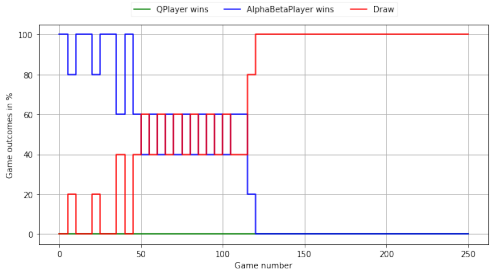

Figure 4: We can see how the player slowly learns to achieve a draw. As we have played 50 rounds of 5 battles, the performance can only increase by $\frac{1}{5} = 20\%$

Making the `QPlayer` play against itself yields interesting results: sometimes the two players continue to try to outplay each other.

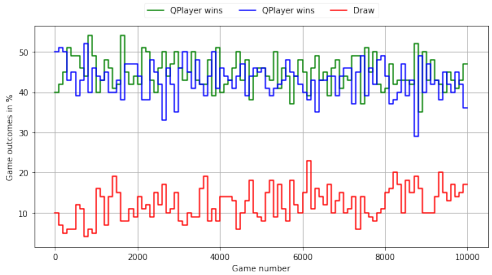

Figure 5: We can see how the player slowly learns to achieve a draw. As we have played 50 rounds of 5 battles, the performance can only increase by $\frac{1}{5} = 20\%$

Sadly teaching the neural network proved to be a tough task. The training procedure takes a very long time, so quick testing is impossible and tuning the hyperparameters proved very difficult, especially within the short period of the course. More optimizers should be considered, as well as different reward strategies. Due to the lack of results for the neural network the reader is invited to review the source code.

References

[1] T. Herlau, *02465 Sequential Decision-Making*. August 17, 2020.

[2] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. The MIT Press, second ed., 2018. http://incompleteideas.net/book/the-book-2nd.html.

[3] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, *Human-level control through deep reinforcement learning Nature* **518** no. 7540, (Feb., 2015) 529–533. http://dx.doi.org/10.1038/nature14236.