Monte-Carlo Tree search for PAC-MAN

Yana Hasson
yana.hasson@student.ecp.fr

Louis Martin
louis.martin@student.ecp.fr

Abstract

Inspired from behavioral psychology, reinforcement learning has been a very active domain in the past few years. The core idea is to make agents learn how to act in a specific situation by encouraging them with rewards. Recent major successes such as the victory of Alpha Go program against the world champion at the game of go [1] emphasize the potential of these approaches.

In this report, we will present our work on the PAC-MAN game using the Monte-Carlo Tree search algorithm. The algorithm that was at the base of Alpha Go. We work on a simplified version of the PAC-MAN, progressively complexifying the rules to check the adaptivity of our agent to incrementally complexified environments.

1 Introduction

Reinforcement learning has been applied to games for a long time because games provide a simplified controllable environment. The first well known successful end-to-end algorithm is TD-Gammon using temporal differences [2]. It is one of the first algorithm to achieve Super-human performance as soon as 1995.

Recent advances in computational power and the revival of neural networks in 2012 since [3] led to successes on more complex games such as the Atari 2600, by using raw pixels only [4]. It even beat humans in most of the games in 2015 [5].

In classic game theory, the game can often be modeled by a tree where each node represent a state, and its children represent the states which can be reach by executing a legal move. Designing a good agent in this case often consists in using a tree search algorithm to find the best nodes. Classic algorithms include Minimax search with $\alpha\beta$ pruning, A^*, etc. However these algorithms often suffer from a high branching factor in games such as the game of Go or Chess in a lesser measure. These algorithms indeed often rely on a complete branch exploration (minus the pruning) which prevents it from exploring very deep in the tree.

Monte Carlo Tree Search on the other hand, uses a Monte Carlo (random) approach for exploring the tree and does not need every state to be evaluated although a good heuristic can be helpful for efficient exploration. This aspect make MCTS a good fit for complex games such as the game of Go [1].

Our implementation is available on Github at https://github.com/louismartin/pac-man.

2 The environment

2.1 The PAC-MAN game

We chose PAC-MAN as our environment because of its fame and its relative complexity which provided an interesting challenge.

The agent is the yellow object at the bottom of figure 1, the Pac-Man. During the game, Pac-Man needs to eat all the little dots called pac-dots without being eaten by the other agents, the ghosts.

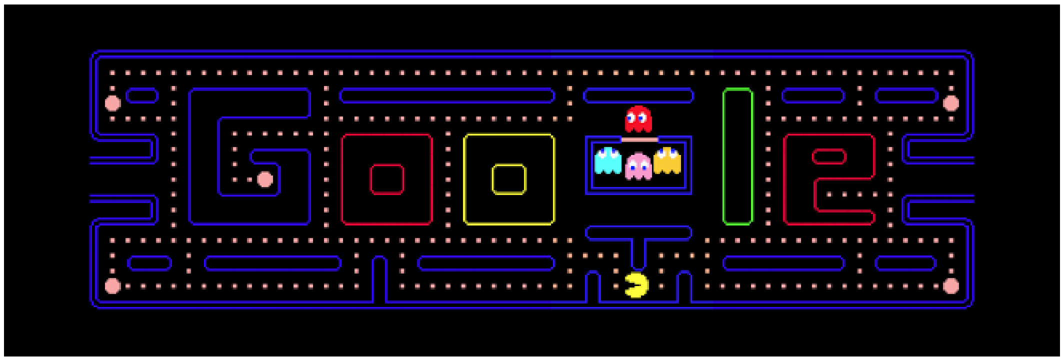


Figure 1: The classical game UI of PAC-MAN

The real Pac-Man game is pretty complex, with the ghosts having semi-random behaviours.

Instead of working on the full set of rules, we decided to create simplified versions of this game keeping the fundamental interaction rules between PAC-MAN, the pac-dots and, when present, the ghosts.

We appreciated the game because there are several ways to increase the game's difficulty, by incrementally adding complexity to the rules.

We can add complexity in several ways:

- increasing the size of the grid
- adding ghosts
- adding ghost candies, which make the ghosts vulnerable for a limited time
- complexify the ghost's behaviour

2.2 Our implementation of PAC-MAN

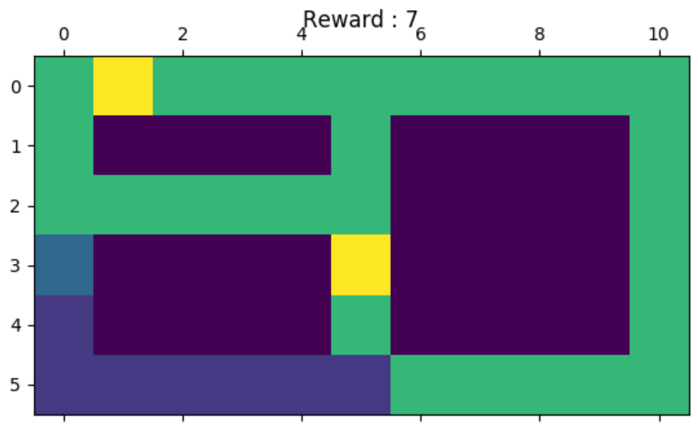


Figure 2: Our implementation using nodes on a graph and matplotlib. We can see the pac-man (blue), the ghosts (yellow) and the pac-dots (green). Purple areas are walls.

We implemented the game environment as a graph. Each node is cell of the grid which is linked to its neighbours in the board.

The agents can then wander around on the board as illustrated in figure 2.

Rewards or candies can be attached to pac-dots. Also, a final reward can be added for managing to win the game : eating all the pac-dots before being eaten by a ghost.

In our basic implementation, each ghost wanders randomly around the board, and the only learning agent is the PAC-MAN.

At each time step the agents can move in the directions that are not occluded by walls. If a collision occurs (pac man and ghost moving towards the same position or towards each-other), the next state of the game is resolved according to the rules.

The end of the game is triggered by the collision between the PAC-MAN and an active ghost or when the PAC-MAN eats all pac-dots.

3 Monte-Carlo Tree Search

3.1 The algorithm

The Monte-Carlo Tree search algorithm is a reinforcement algorithm which models the state space as a tree, introduced by Coulomb et al in 2006 [6]. In order to find the best action at a specific state, we need to choose the best child for the current tree node. It uses a Monte-Carlo approach for tree exploration, i.e. sampling random trajectories that explore the most promising nodes in priority. This allows efficient exploration without computing the absolute best choice at each step.

The algorithm is divided in four stages illustrated in figure 3.

- **Selection**:
  Start from the root and for every node that was already visited, apply a selection strategy to choose the next node. The selection strategy controls the balance between exploration and exploitation. It can be interpreted as a multi-armed bandit problem at each node, we can therefore use algorithms such as UCB.
- **Expansion**:
  When reaching the end of the tree, we select a new node to be expanded and add it to the tree. Other variants add nodes up to a certain depth during the expansion step, however adding too many nodes may consume too much memory.
- **Simulation**:
  At this stage we only face node that were not explored yet. The exploration follows a predefined simulation strategy until the end of the game
- **Backpropagation**:
  Now that we have the final result of the game we just played, we can backpropagate it through all nodes that were visited. Let's say that at each node we keep track of the number of wins and losses after going through that node. If the game is won, the backpropagation stage would consist in adding 1 to the win count of every visited node.

3.2 Our implementation of MCTS

We implemented the tree using a python dictionary for its fast hash-table underlying structure. This approach provides O(1) membership testing to check if a node was already visited and O(1) element retrieval for backpropagation.

Each node is uniquely defined by its key in the dictionary, which is a hashable representation of the state. The key indeed needs to be a hashable data type in order to be used in the dictionary, this is why we represented the states with a tuple containing the features.

Each node is an object storing the cumulative reward and the number of visits during selection phase.

The tree is therefore not aware of its own structure, i.e. it does not know how the nodes are linked. This information is given by the environment through legal actions. The environment indeed decides which states are accessible from a given state.

One particularity of our environment is that the game can go through a same states multiple times in the same game. The tree is therefore not properly speaking a tree but a graph.

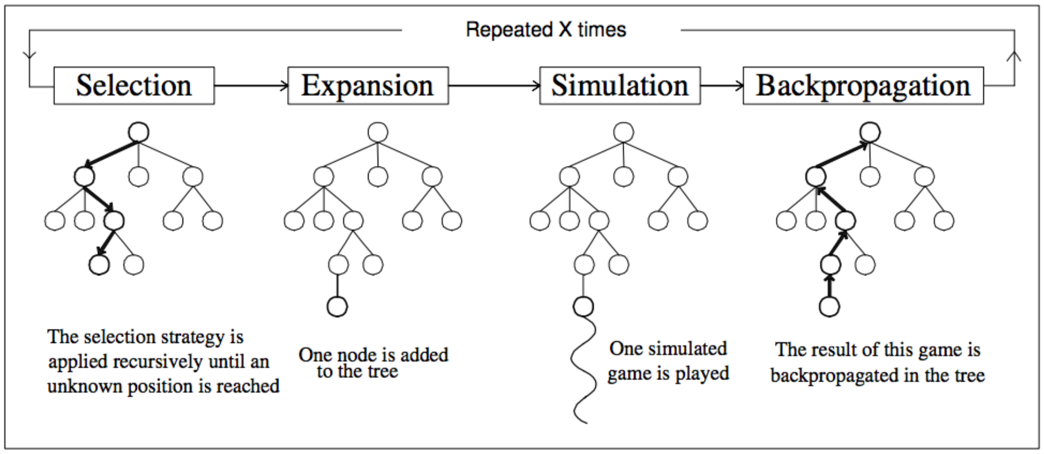

Repeated X times
Selection
Expansion
Simulation
Backpropagation
The selection strategy is applied recursively until an unknown position is reached
One node is added to the tree
One simulated game is played
The result of this game is backpropagated in the tree

Figure 3: Monte-Carlo Tree Search outline from [7]

This brings the problem of loops, sometimes the agent would loop indefinitely during the selection or simulation stage because of the graph structure and the fact that the results are only backpropagated until the end.

That is why we set a timeout during training after which the game is lost (which also seems natural, because we don't want the agent to take ages to win). The timeout can be removed without any negative consequence when the probability of an encounter with a ghost is high and therefore the game ends in a limited number of steps.

Another problem occurs when backpropagating the results. As a single state may have been visited multiple times, we need to be careful to update the results for this state only once.

Bandit algorithm

- **UCB**:
  In our implementation we first used the Upper Confidence Bound (UCB) algorithm and applied it to Tree search. We select the next node by choosing the one that maximizes the UCB_score.

  $$UCB\_score(node) = \mu(node) + \sqrt{\frac{log(time\_step)}{(2 * number\_visits(node))}}$$

  Where number_visits is the number of time the node was visited, time_step is the index of the simulation step, and μ is the empirical mean of the rewards that were ultimately obtained when choosing this node.

- **KL-UCB**:
  The KL-UCB algorithm was introduced by [8] in 2011. It has better theoretical guarantees than UCB and performs better in practice too. It is based on maximizing the Kullback-Leibler divergence in order to be as close as possible to the upper-confidence bound.

  We want to draw the arm that maximizes the KL_UCB_score. This score is computed for arm a as follows:

  $$KL\_UCB\_score = \max\left(q \in [0,1] : N[a]d(\hat{\mu}(a), q) \leq log(t) + clog(log(t))\right)$$

  with $\hat{\mu}(a)$ the arm's empirical mean, $N[a]$ the number of time the arm was drawn, c a parameter (set to 0 in our code as advised in the paper) and $d(p, q)$ the Kullback-Leibler divergence of the Bernouilli distribution.

  Although the former is computed for the Bernouilli distribution it is shown that it can be used for any distribution given that the rewards are between 0 and 1. We therefore have to rescale the final cumulative rewards of the game to $[0, 1]$ in the code.

We have

$$d(p,q) = p \log \frac{p}{q} + (1-p) \log \frac{(1-p)}{(1-q)}$$

. and

$$\frac{\partial d}{\partial q} = \frac{q-p}{q(1-q)}$$

. As is said in the paper, for $p \in [0,1]$, $q \mapsto d(p,q)$ is strictly convex and increasing on $[p,1]$ (clearly $\frac{\partial d}{\partial q} > 0$ for $q \in [p,1]$). We can therefore use Newton-Raphson optimization method to find the best q.
We implemented KL-UCB and showed that it outperformed UCB on a simple multi-arm bandit problem with 10 bernouilli arms on figure 4. We will compare the two algorithms used in MCTS in the following section.

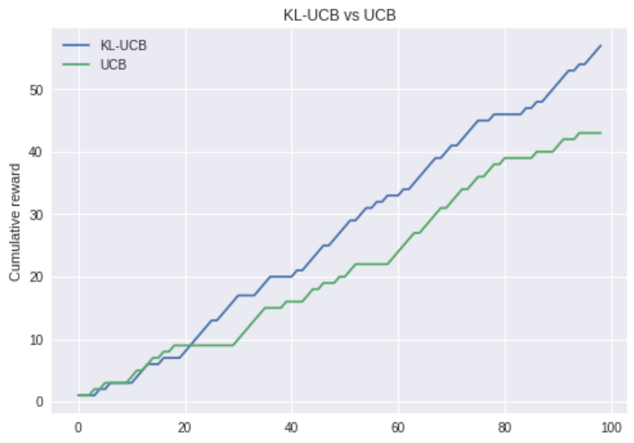


Figure 4: KL-UCB vs UCB on a simple multi-arm bandit problem with 10 bernouilli arms

4 Results

4.1 Workflow

We experimented first on small grids, in order to check the correctness of our method's implementation.

As our ultimate goal is to be able to solve the problem for larger grids, we considered approaches that have the potential to handle larger grids. Although the number of moves for the PAC-MAN at each moment is limited, for larger grids, the number of possible states is very large due to variety of possible positions for the pacman and the boards's setting.

Approaches that rely on a full search of the game tree, such as minimax for instance are not adapted to such large state spaces.

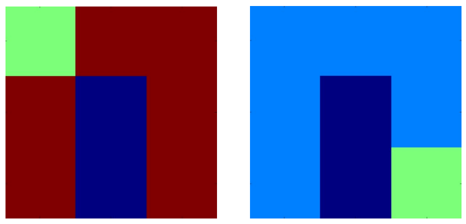
That is why we implemented Monte Carlo Tree Search as described in [7].

4.2 Experiments

3x3 Simplistic grid The very first grid we experimented with was a 3*3 grid as presented in 5 with 6 pac-dots to be eaten and no ghosts.

To avoid that the agent would indefinitely walk the grid without eating all the pac-dots during the early stages of the training, we artificially ended the game after 100 steps.

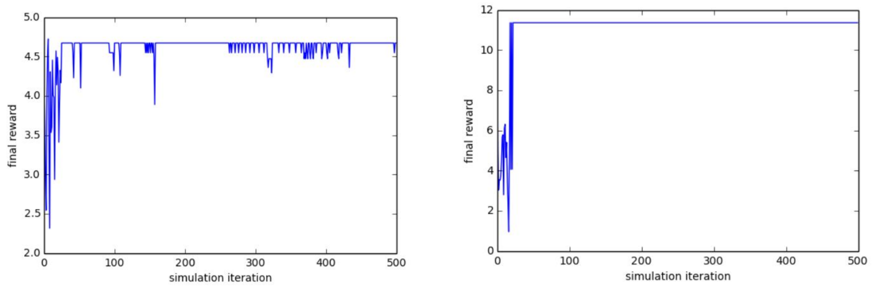
We found that the MTCS algorithm worked pretty well for a this simple grid, with a discount factor of 0.95 for the reward at each step, the agent often converged to good solutions, eating all the candies on one side and then all the ones on the other side of it's initial position.



(a) initial board (b) game won

Figure 5: 3x3 simplistic grid, pacman in green, pacdots in red

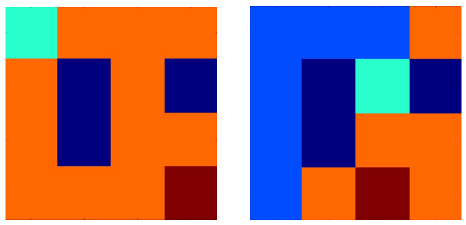
We observed, as presented in 6 that adding a winning reward when finishing the game tended to favor a convergence towards a stable solution.



(a) winning reward : 0 (b) winning reward : 10

Figure 6: final rewards during training simulations

4x4 grid We then moved on towards a larger grid of size 4x4 and added a ghost to the grid as displayed in 7.



(a) initial board (b) ongoing game

Figure 7: 4x4 grid, pacman in green, ghost in red, pacdots in orange

In this case, because of the random behavior of the ghost, the pacman did not converge to a given solution but training still occurred. To visualize the training we used a running-mean that averages the results over a given mean-step.

In this case, the discount-factor proved to be a critical parameter. To display the effect of this parameter, we trained the agent for various discount factors for a fixed time of 100 seconds and display some training statistics in 8. As the probability of meeting the ghost is relatively high, decreasing and finally removing the discount factor actually increases the value towards which the final reward converges. As there are 11 pacdots on the grid, we see that the solution without discount factor manages to get a high reward in most cases after an initial period of training. We must notice that changing the discount factor affects the final reward : the maximal reward that can be gained in the game is strictly decreasing with the decrease of the discount factor. Therefore, to be able to compare the performances of the algorithms with different discount factors, we chose to also monitor the number of wins in a game. This gives an indirect measure of the performance of the algorithm but it has the advantage of allowing us to compare all the algorithms on a comparable basis.

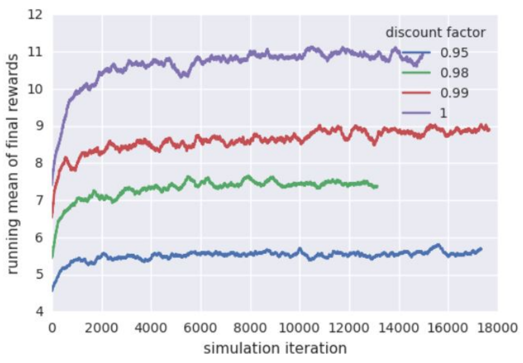
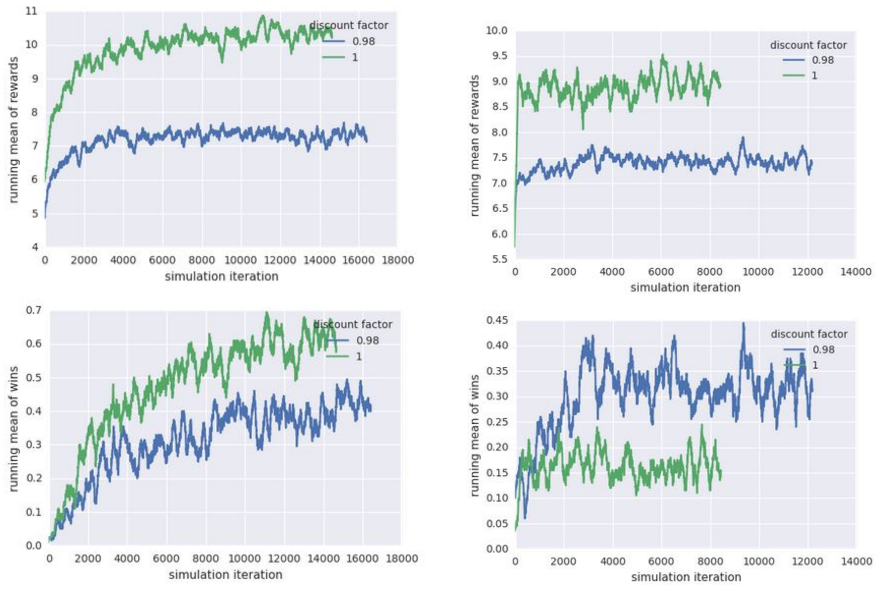


Figure 8: running mean of final rewards (mean-step : 200)

In order to prepare for further increase of the board, we introduce hand-crafted features. Instead of training the agent on the visual representation of the board, we give it the following features as state:

- relative coordinates of the closest ghost
- relative coordinates of the closest pac-dot

We present the performances for the two types of state representations in 9. We can see that the simple hand-crafted features performance is worse than the board representation, both in terms of final reward and number of wins, but we can also see that providing hand-crafted features speeds up the training in the very early stages of the training.



(a) board as state (b) hand-crafted features

Figure 9: 4x4 performance during training simulations

We also compared UCB and KL-UCB for the best-performing algorithm (discount factor of 1 and entire board as state).

We see looking at 10 that standard UCB performs better than KL-UCB with MCTS as UCB learns faster to obtain high rewards, but that both manage to reach the same level of average reward.

6x11 grid We further expanded the size of the grid and to reach the board presented in 2. At first we experiment a with **a single ghost** on the board.

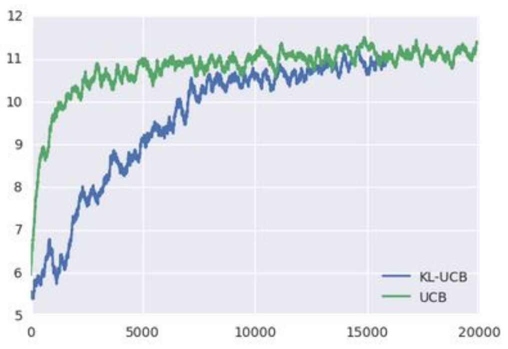
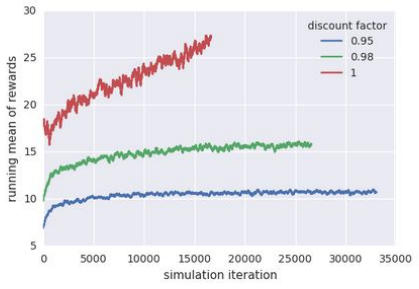
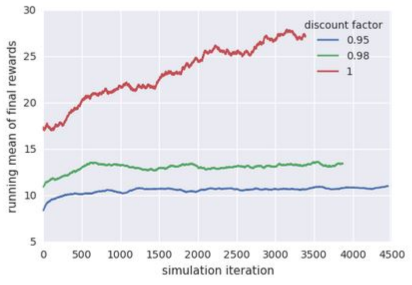


Figure 10: running mean of final rewards on 4x4 board (running mean step : 200)

We compare the board state and the hand-crafted features in this configuration. 11. We can see the benefit of using hand-crafted features.



(a) board as state



(b) hand-crafted features

Figure 11: 6x11 with 2 ghost performance during training simulations

As in the case of no discount we see that the training did not stabilize. We therefore continue the training for a longer time (two and a half hours) in the pixel state space. We see in 12 that the algorithm manages to learn how to play well enough to win a majority of games.

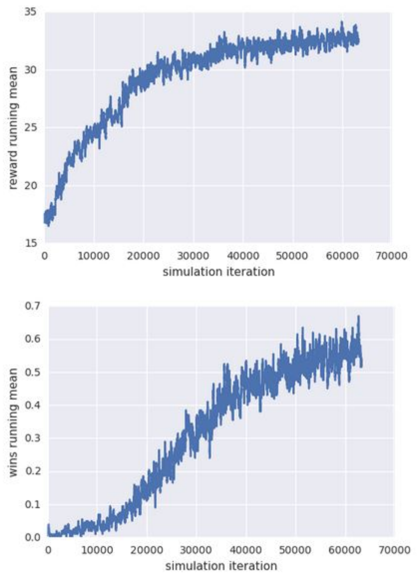


Figure 12: performances for 6x11 board with one ghost (mean-step : 200, discount-factor : 1)

We also compared UCB and KL-UCB in this configuration. As we can see in 13, in the early stages UCB and KL-UCB perform comparatively. In the later stages, UCB outperforms KL-UCB.

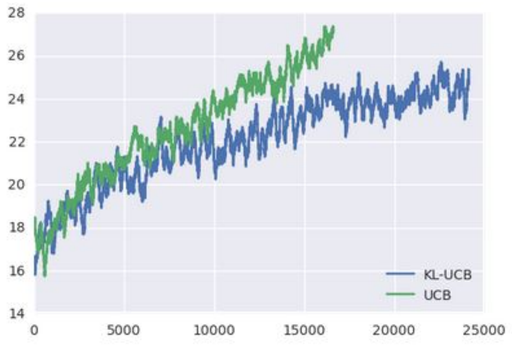
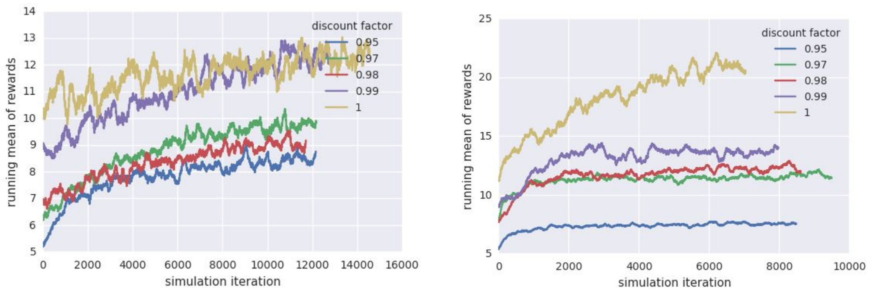


Figure 13: performances for 6x11 board with one ghost (mean-step : 200, discount-factor : 1)

As a last addition to complexity, we decided to also make the game more difficult by adding **a second ghost**.

In this configuration our implementation of MCTS produced poor results: wins become scarce. We see that in this configuration,learning using the entire pixel space becomes difficult and using hand-crafted features proves to be relevant 14.



(a) board as state

(b) hand-crafted features

Figure 14: 6x11 with 2 ghost performance during training simulations

To compare the performance of the various discount factors we looked at the total number of wins over the training period and found that no discount produced the highest proportion of wins (29 wins out of 7000 simulations).

We see in 14 that hand-crafted features perform much better than the entire board representation. In a smaller number of iterations we manage to obtain higher reward scores for most discount factors, and especially for the best configuration without any discount factor.

5 Limits and improvements

Improvement of the simulation step In the simulation step, it is possible that we meet a state that has been visited before. In our implementation, we do not take advantage of the knowledge we have on this given step and continue to randomly select steps until the final step is reached. Instead, we could have systematically checked if we have information about the current node that allows us to choose the next node wisely.

The problem with this potential improvement is that the backpropagation step would become more tricky as we would have to choose for the nodes a new criterion to decide whether we update their score.

Generalization limits When pacman is trained on the board directly, it learns to deal with a specific configuration of the board. When presented with a new board, it has to learn the good moves from scratch. On the other hand, when using hand-crafted features, pacman can be board-agnostic and therefore reuse previously gathered knowledge from smaller or different boards.

Feature engineering Here we only experimented with basic feature engineering, we saw that this proved useful for larger and more complex grids. We could further experiment with feature engineering by providing more complex features. We could for instance add some local knowledge by providing a restricted view of the board centered around the PAC-MAN agent. Also, in our implementation we did not take advantage of the fact that coordinates are ordered integers, we could therefore try to use this knowledge to find best actions on never-seen-before states by finding the closest corresponding state or weighing various contributions by proximity.

6 Conclusion

We successfully showed that Monte-Carlo Tree search could be used to learn how to play PAC-MAN in simple configurations. We first experimented on small grids (3x3 and 4x4) and then added complexity with bigger grids and additional ghosts.To handle this additional complexity, we created tailor-made features to reduce the state space size. When we added a second ghost on the 6x11, the state space size degraded the performances of the pixel state-based algorithms. In this case, hand-crafted features were associated with faster learning and better performances. We also compared UCB with KL-UCB, and although on simple board they reach the same performance asymptotically, UCB is faster to converge than KL-UCB. On more complex boards, UCB outperforms KL-UCB as well. Overall, we saw that given enough time, and a limited number of ghosts, giving the entire board as state allowed the agent to learn efficiently how to win the game. To handle larger grids, additional engineering is required.

References

[1] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, January 2016.

[2] Gerald Tesauro. Temporal difference learning and td-gammon. *Communications of the ACM*, 38(3):58–68, 1995.

[3] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[4] Marc G Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 2012.

[5] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[6] Rémi Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In *International Conference on Computers and Games*, pages 72–83. Springer, 2006.

[7] Guillaume Chaslot. Monte-carlo tree search. *Maastricht: Universiteit Maastricht*, 2010.

[8] Aurélien Garivier and Olivier Cappé. The kl-ucb algorithm for bounded stochastic bandits and beyond. In *COLT*, pages 359–376, 2011.