Knight vs King Chess with Reinforcement Learning

Marco Solime
Master's Degree in Artificial Intelligence
University of Bologna
`marco.solime@studio.unibo.it`

Abstract

In this project, an agent is trained to solve a chess-inspired puzzle game, where a knight must check the opponent king in the least amount of moves without being captured. To do this, we provide two approaches based on Reinforcement Learning, leveraging Deep Q-Networks and Actor-Critic approaches. We show that while both methods share similarities and significantly outperform a random baseline, they differ under more subtle aspects.

1 Introduction

Board games have become the perfect test bed to evaluate Reinforcement Learning (RL) algorithms, due to their high number of states, search space, and combinatorial nature. While traditional algorithms used techniques based on brute force search and pattern matching (e.g. DeepBlue [2]), modern approaches leverage RL, deep neural networks and self-play to achieve superhuman performances, as shown in AlphaGo Zero [3]. While building a sophisticated chess program usually requires days of training, dedicated hardware and parallel computing, we constrain ourselves to a simpler version of chess, where only 2 pieces appear on the board, and try to solve it using RL algorithms.

2 Game Description

2.1 Game Rules

The game takes place on a 8x8 chessboard. There are only two pieces on the board: a black knight and white king. The knight is the only movable piece on the board, while the king remains in the same position for the whole game. The knight moves in the so-called L pattern: 2 squares in any direction, 1 square to left or right. It can jump over pieces that may be on its way. Starting from a random legal position, the black knight aims to check the white king in the least amount of moves. The knight loses the game if it is captured by the king - i.e. it lands on a neighboring square of the king. The knight has a total of 10 moves to find a way to check the opponent king. If no such event occurs, the knight runs into an Out Of Moves (OOM) loss.

2.2 Custom Environment

To let the agent learn through experience and improve its policy, we need to build a custom environment. It must be able to give positive feedback for correct actions and provide negative outcomes for mistakes. In this way, our hope is that the agent will learn a good policy by interacting with the environment, observing states, taking actions, and receiving rewards. In particular, we need to define the observation state, the action space, the reward mechanism, and the game logic. Regarding the latter, we need to make sure the knight only performs legal moves and prevent him to land on squares outside of the board.

Preprint. Under review.

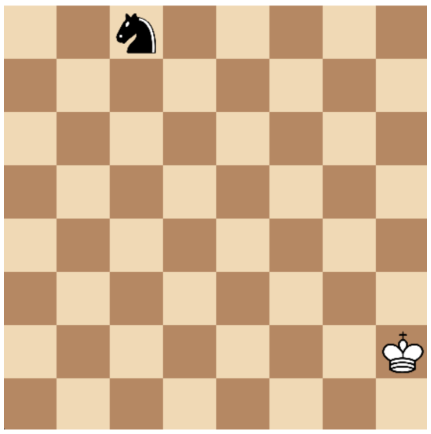

Figure 1: A screenshot of the game.

Observation Space The observation space is described by 4 discrete numbers: x and y coordinates of the knight, x and y coordinates of the king. Each number ranges from 0 to 7. Using OpenAI Gym [1] format, we create an object of type

Box(low=0, high=7, shape=(4,), int32)

Regarding the observation space, other methods are possible.

- Matrix representation: an 8x8 matrix where each cell can take 3 numbers: 0 is empty, 1 is knight, 2 is king.
- Frozen Lake-style [1]: 2 discrete numbers that describe the "unrolled" absolute position of the pieces.

Our solution is a midway of the two: while being substantially lighter than the 8x8 version, it provides more structure than the "unrolled" strategy. The number of the states is $\sim 8^4 = 4096$. The actual number is slightly lower than 4096 because a small fraction of states correspond to illegal positions, e.g. the knight checks the king at the first move, or the knight overlaps with the king.

In general, the matrix representation scheme is preferable because it can be easily extended to games with more pieces on the board, e.g. Chess, Checkers, Go. Since our game involves only 2 pieces, a simpler representation is chosen.

Action Space The action space is defined by a discrete number. Indeed, a knight has 8 possible moves, and each of them is numbered from 0 to 7. Using OpenAI Gym notation, we create an object of type

Discrete(8)

Rewards Regarding the reward system, we devise the following scheme

a. -1: neutral move
b. -5: illegal move (knight moves out of board)
c. -5: knight out of moves (OOM)
d. -10: knight captured (king wins)
e. +10: king checked (knight wins)

(a) To encourage the knight to check the king in the least amount of moves, we penalize with a small negative reward each move. This approach is normally performed on maze-like games. The idea is that a faster path to the target is preferable to a path with loops or repeated moves. (b) We also give a negative reward for illegal moves, namely moves that land outside of the board. In this way we hope our agent becomes aware of the feasible moves while not directly telling him. (c) We also penalize the agent when it exceeds the move limit (OOM). (d) We give a high negative reward for capturing moves. In this state, the game ends with a loss for the knight. (e) We give a high positive reward when the knight checks the king. Here we end the episode with a win for the knight.

3 System Description

3.1 Deep Q-Learning

The first approach is based on Deep Q-Learning, a.k.a Semi-gradient Q-Learning. The method combines Q-learning with neural networks. The deep neural network consists of a stack of linear layers with ReLUs in between, the dimensionality of the hidden dimension being 128. The network takes in 4 floats (game state) and outputs 8 values corresponding to the knight's moves. The update rule of the network parameters is as follows:

$$w_{t+1} = w_t + \eta \left[R_{t+1} + \gamma \max_a \hat{q}(S_{t+1}, a, w_t) - \hat{q}(S_t, A_t, w_t) \right] \nabla \hat{q}(S_t, A_t, w_t) \tag{1}$$

where w_t is the vector of the network's weights, η is the learning rate, R_{t+1} is the immediate reward, γ is the discount factor, A_t is the action selected at time t, and S_t and S_{t+1} are respectively the inputs to the network at time steps t and $t+1$.

The loss minimizes the difference between the predicted Q-values and the target Q-values. $\hat{q}(S_t, A_t, w_t)$ is the Q-value predicted by the neural network parametrized by w at time t. The target Q-value for a state-action pair (s,a) is given by $y = R_{t+1} + \gamma \max_a \hat{q}(S_{t+1}, a, w_t)$. Therefore, the loss function can be written as the mean squared error between the predicted Q-values and the target Q-values.

$$L(w_{t+1}) = \mathbb{E}_{(S_t, A_t, R_{t+1}, S_{t+1}, done_{t+1}) \sim D} \left[(y - \hat{q}(S_t, A_t, w_t))^2 \right] \tag{2}$$

D is the experience replay buffer containing tuples accumulated through game-play.

Epsilon-greedy To balance exploration and exploitation, we used an epsilon-greedy strategy. The idea is that at the beginning the agent explores more (high epsilon), and over time, it starts exploiting the learned knowledge (low epsilon). Throughout the training phase we always keep a minimum level of exploration to avoid getting stuck in suboptimal policies.

In particular, we experimented with 3 decay schedules.

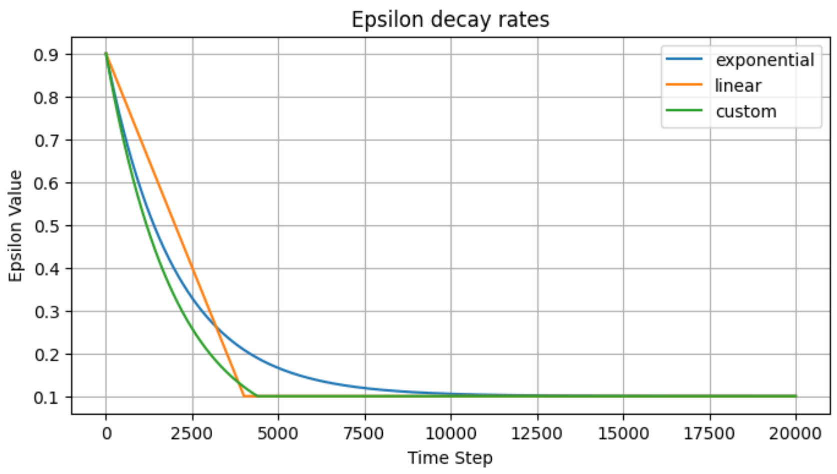

Figure 2: Epsilon decay rates plotted on 20K episodes.

$$\epsilon(t) = \epsilon_{min} + (\epsilon_{max} - \epsilon_{min}) \cdot e^{-kt} \tag{3}$$

$$\epsilon(t) = \max(\epsilon_{min}, \epsilon_{max} - kt) \tag{4}$$

$$\epsilon(t) = \begin{cases} \epsilon_{\text{max}} \cdot \beta^t & \text{if } \epsilon_{\text{max}} \cdot \beta^t > \epsilon_{\text{min}} \\ \epsilon_{\text{min}} & \text{otherwise} \end{cases} \tag{5}$$

Equation (3) uses an exponential decay strategy. Starting from ϵ_{max} (close to 1) it exponentially decays till a minimum value ϵ_{min}. We adjust the decay rate through k, a positive constant. In equation (4), epsilon decays linearly from ϵ_{max} to ϵ_{min}, maintaining ϵ_{min} throughout all the remaining episodes. In equation (5) it is shown an hybrid method, where epsilon is exponentially decayed till a ϵ_{min}, before being truncated to ϵ_{min}. β_t is the decay factor (e.g. 0.99). Decay rates are shown in Figure 2. We find that the the hybrid (custom) strategy works best in our experiments.

Experience Replay Buffer We made use of experience replay to deal with the convergence problem of Q-learning. Indeed, we know that highly correlated updates lead to slow convergence speed. To mitigate this issue, we set up a replay buffer of length 1000, storing tuples of the following shape.

- S_t: the current state
- A_t: the executed action
- R_{t+1}: the returned reward
- S_{t+1}: the next state
- $done_{t+1}$: a boolean indicating whether the episode is terminated

After each episode, we sample from the replay buffer a minibatch of tuples (64), and update the DQN. We implement it using a double-ended queue. This means that once the buffer is full, we start appending new elements to it, overwriting old ones. We sample from the buffer using a uniform distribution.

3.2 One-Step Actor-Critic

The second approach is based on the Policy Approximation method. In this scenario we have 2 neural networks: an actor and a critic. The actor produces a probability distribution over all the possible actions; the critic outputs a single scalar value, denoting the evaluation of the state. Both networks consist of 3 linear layers with ReLUs in between, the hidden dimension having dimensionality 128. The output activations of the actor are passed through a softmax activation function to obtain probabilities. To take an action, we sample from the output probability of the actor, and provide such action to the game environment. Both actor and critic are updated on-policy using the current state-action-reward transitions.

Critic Update It involves computing a temporal difference error (TD error) delta, which is used to update the parameters of the value function.

$$\delta = R_{t+1} + \hat{v}(S_{t+1}, w_t) - \hat{v}(S_t, w_t) \tag{6}$$

R_{t+1} is the immediate reward, $\hat{v}(S_{t+1}, w_t)$ is the estimated value of the next state S_{t+1} using the current critic parameters w_t, $\hat{v}(S_t, w_t)$ is the estimated value of the current state S_t using the current critic parameters w_t. Therefore, the critic parameter update becomes:

$$w_{t+1} \leftarrow w_t + \alpha_{critic} \delta \nabla \hat{v}(S_t, w_t) \tag{7}$$

where α_{critic} is the critic's learning rate, $\nabla \hat{v}(S_t, w_t)$ is the gradient of the value function with respect to its parameters w_t. The loss function can be considered as the Mean Squared Error (MSE) between the TD target $R_{t+1} + \hat{v}(S_{t+1}, w_t)$ and the current value estimate $\hat{v}(S_t, w_t)$.

$$L_{critic} = \frac{1}{2}\left(R_{t+1} + \hat{v}(S_{t+1}, w_t) - \hat{v}(S_t, w_t)\right)^2 \tag{8}$$

This loss is then minimized to reduce the difference between the predicted value and the TD target.

Actor update It is based on the policy gradient theorem and uses the TD error delta as an estimate of the advantage.

$$\theta_{t+1} \leftarrow \theta_t + \alpha_{actor}\delta\nabla \log \pi(A_t|S_t, \theta_t) \tag{9}$$

α_{actor} is the learning rate, $\nabla \log \pi(A_t|S_t, \theta_t)$ is the gradient of the log probability of the taken action A_t under the current policy π with respect to the policy parameters θ_t. Hence, the loss function can be considered as the negative log-probability of the taken action weighted by the TD error.

$$L_{actor} = -\delta \log \pi(A_t|S_t, \theta_t) \tag{10}$$

This loss is minimized to increase the probability of actions that lead to higher returns.

4 Experimental setup and results

In order to fully assess the performances of our models and estimate how they compare to each other, we devised the following metrics. All metrics are averages and are computed after a sufficient amount of episodes (i.e. 20K games).

- cumulative reward
- number of legal moves
- number of illegal moves
- winning probability
- exceeding max move length probability

Our approaches are also evaluated against a random baseline - an agent that moves the knight uniformly at random. While training our models, we also trace the cumulative average reward for each episode.

As shown in Figure 3 Both DQN and A-C significantly outperform the Random baseline in terms of average cumulative reward over the episodes. Before episode $\sim$5000, AC is slightly preferable then DQN. However, from $\sim$5000th episode, DQN clearly surpasses AC and keeps a neat margin for the rest of the games.

Evaluating the agents on 1000 games, we can draw the following conclusions. Results are shown in Table 1. (A) In terms of average cumulative reward, DQN and AC seem to be close to each other, despite DQN slightly outperforming AC. (B) On the contrary, AC takes on average less moves to complete a game, differently from DQN. (C) AC is also more robust than DQN in terms of illegal moves, since it rarely performs any out of the board move. (D) When it comes to chance of winning, DQN slightly beats AC. (E) Interestingly, AC is less prone to run into out of moves games. We believe this is a consequence of training from trajectories of moves instead of sampling uncorrelated examples from the experience replay buffer.

Table 1: Performance metrics as averages computed on 1000 games.

Model	Cum. Reward	Legal Moves	Illegal Moves	% Win	% OOM
DQN	**-0.052**	4.46	0.217	**0.642**	0.313
AC	-0.589	**3.444**	**0.002**	0.579	**0.057**
Random	-17.577	4.185	2.484	0.249	0.468

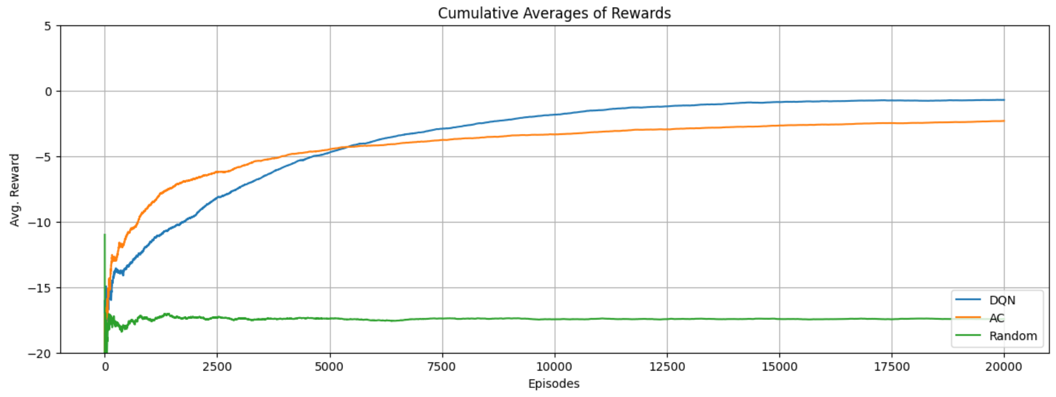

Figure 3: Average cumulative rewards over the played games for DQN, A-C, and the Random player.

5 Discussion

DQN has a higher chance of winning the game compared to AC. Empirically, it shows complex knight maneuvers, bringing the knight closer and closer to the king move after move. Additionally, DQN rarely leaves the knight under king's attack. Such considerations are reflected on the average cumulative reward and winning probability, both higher than AC's (Table 1). On the other hand, AC generally takes less moves to win the game. Moreover, AC seems to have gained knowledge of illegal moves.

From a practical viewpoint, DQN is prone to run into loops. Such repetitions often happen when the king is in the corner or on the edge of the board. In these circumstances, the knight has few candidate squares to check the king. (In the case of the corner, there are only 2 check moves). Therefore, finding check moves becomes more challenging, as it is generally needed to do a look-ahead of more than 3 moves. The loops performed with DQN usually involve 2 (back-and-forth), 3 (triangles), and 4 squares (diamonds) patterns. We also notice that the knight may get stuck into illegal moves, repeating them indefinitely, running into out-of-moves situations. It is believed that since DQN at test time uses a greedy approach, selecting always the best action at a particular state, there is no way to get out from infinite loops. On the contrary, since AC uses a sampling scheme, it almost never runs into loops and effectively learns to end the game sooner. Regarding the latter aspect, learning from consecutive trajectories is more helpful than sampling uncorrelated samples from a replay buffer.

6 Conclusion

We built a customized chess-inspired game where a black knight aims to check a king in the least amount of moves, while trying not to be captured and run out of moves. We designed a custom environment, defined the observation space and action space, and devised a reward scheme. Our agents are based on neural networks, featuring linear layers and ReLU activations. We also implemented and trained two RL strategies - Semi-gradient Q-Learning and One-Step Actor Critic -, exploiting both value approximation and policy gradient methods. We evaluated such methods against each other and a random baseline, and assessed their performances using a number of different metrics. We let the agents play simulated games and we assessed their points of strength and weakness. While being a quite simple environment, the project demonstrates the power of RL techniques applied to board games, and it opens the path to potential applications in other domains.

References

[1] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

[2] Murray Campbell, A.Joseph Hoane, and Feng hsiung Hsu. Deep blue. *Artificial Intelligence*, 134(1):57–83, 2002.

[3] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of Go without human knowledge. *Nature*, 550(7676):354–359, 2017.