ReinForceMate

A simplified version of chess game via RL algorithms

Silvio Baratto
Valeria Insogna
Thomas Verardo

University of Trieste

Data Science and Scientific Computing

September 18, 2023

Presentation Overview

Introduction

Two different tasks in chess:

- **Shortest path**: is a deterministic game whose aim is to find the shortest path between two cells according to the piece (agent) using both RL algorithms
- **Capture pieces**: the goal is to capture as many of the opponent's pieces in *n* moves using RL techniques.

Shortest path

Introduction

Goal

Learn how to find the shortest path between two squares on a chess board depending on the piece.

Why

The environment (board) is discrete with a small state space.

Environment

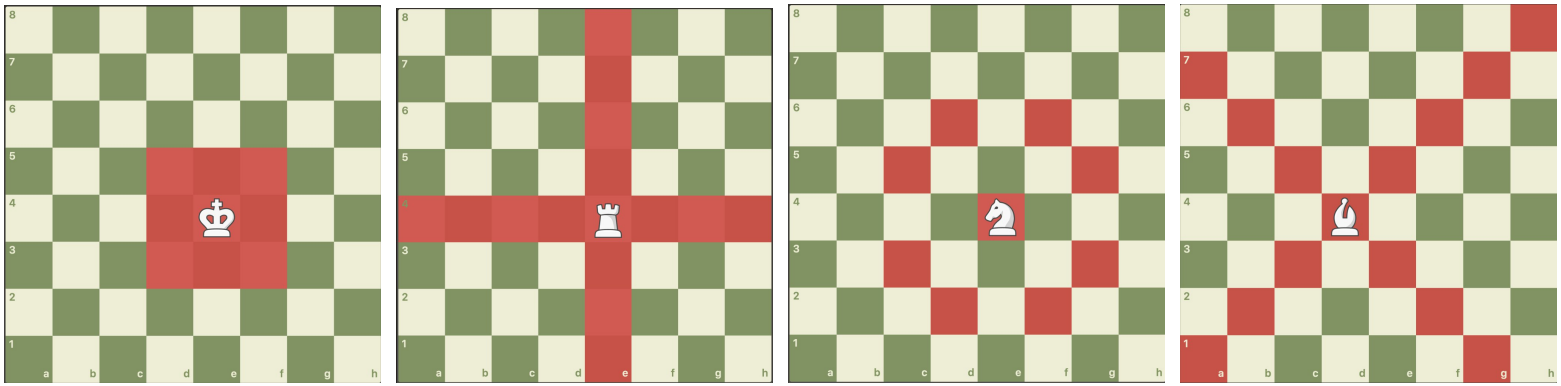
The chess board is represented by 8*x*8 squares, each square represents a state.
The environment is deterministic

- The starting state is **(0,0)** and ending state is **(4,6)**

```
[['S', '', '', '', '', '', '', ''],
 ['', '', '', '', '', '', '', ''],
 ['', '', '', '', '', '', '', ''],
 ['', '', '', '', '', '', '', ''],
 ['', '', '', '', '', '', 'F', ''],
 ['', '', '', '', '', '', '', ''],
 ['', '', '', '', '', '', '', ''],
 ['', '', '', '', '', '', '', '']]
```

Every move from state to state gives a **reward** of -1.
Naturally the best policy for this environment is to move from S to F in the lowest amount of possible moves.

Agent



- 4 different types of agents: king, rook, knight, bishop
- The agent can take a number of actions, which consist of moving a piece to a new square
- Each agent has its own action space.

Shortest path

Markov Decision Process

Since we have a problem composed of actions, states, rewards and transition probabilities, we use MDP to model the problem.

To solve the MDP we use:

- Value iteration
- Policy iteration

The **transition probability** is 1 for all non-terminal states and 0 for the terminal state.

Model based

Policy iteration

We start with an initial value function with 0 for each state and an initial policy that gives an equal probability to each action.

Then we iteratively:

1. Evaluate (repeat until the values converges)

$$V_\pi(s) = \sum_{s' \in s} P_{\pi(s)}(s'|s) * [r(s, a, s') + \gamma v_\pi(s')]$$

2. Improve the policy at each state

$$\pi(s) = argmax_a \sum_{s'} P(s'|s, a) * [r(s, a, s') + \gamma V(s')]$$

Model based

Value iteration

Policy evaluation and policy improvement are combined into a single step.

Starting as policy iteration, find iteratively the optimal value function with:

$$V_\pi(s) = max_a \sum_{s'} P(s'|s,a) * [r(s,a,s') + \gamma V(s')]$$

Model free

Q-learning, SARSA and expected SARSA

- **Q-Learning**

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha[R_{t+1} + \gamma max_a Q(s_{t+1}, a) - Q(s_t, a_t)]$$

- **SARSA**

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha[R_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$$

- **Expected SARSA**

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha[R_{t+1} + \gamma \sum_a \pi(a|s_{t+1}) Q(s_{t+1}, a) - Q(s_t, a_t)]$$

- **SARSA lambda**

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha \delta_t E(s_t, a_t)$$

Results

- Model based techniques have the minimum cumulative reward (-6, the number of steps the reach the final point)
- But they are the worst computational performant

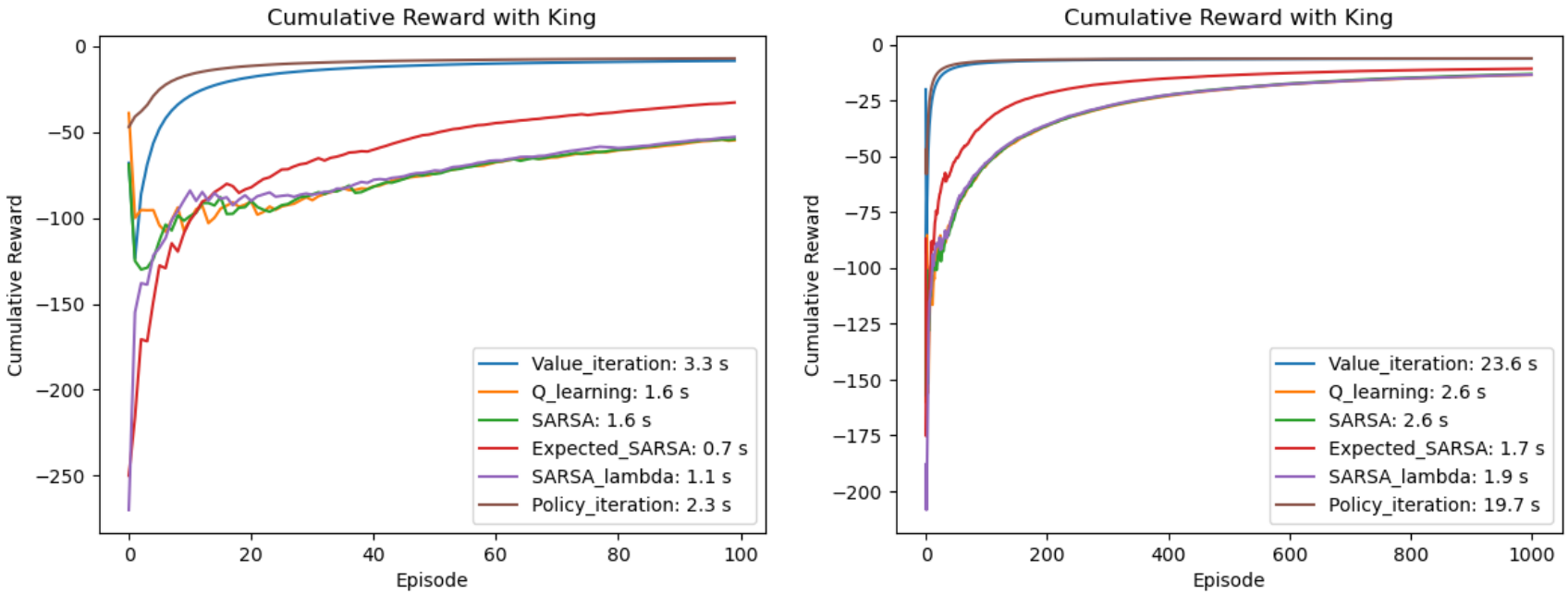
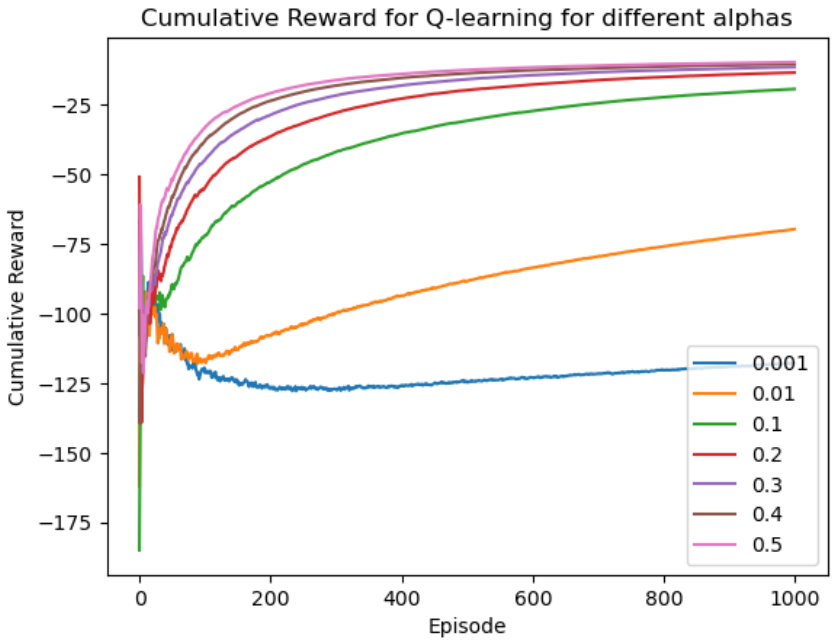
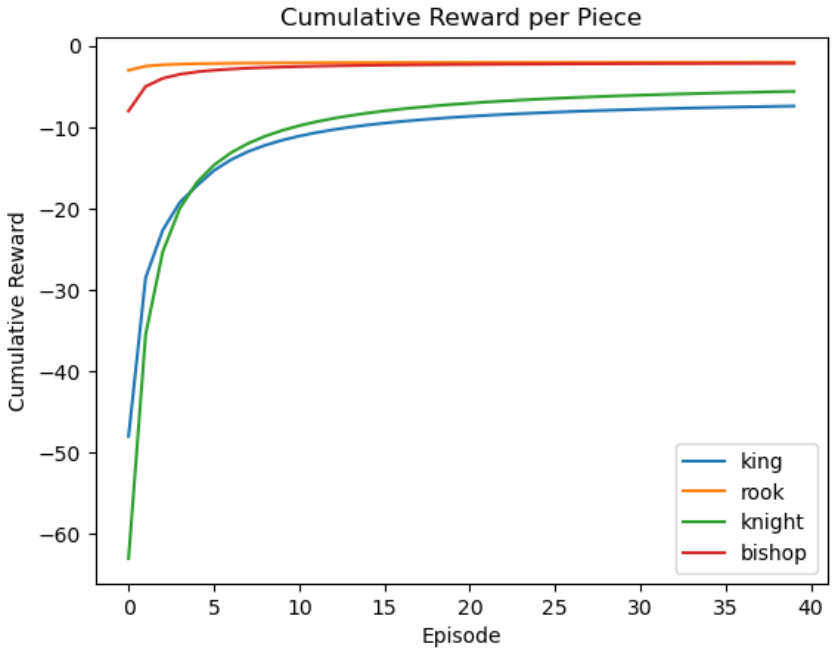


Figure: Average cumulative reward with piece ”King” trained on 100 episodes (left) and on 1000 episodes (right)

Results

We will use $epsilon = max(1/(1+k), 0.05)$ to help the agent to explore more initially and then prioritize exploitation as it becomes more knowledgeable about the environment.

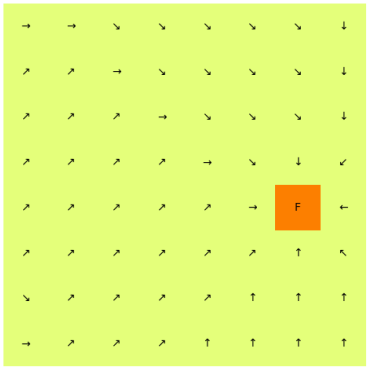




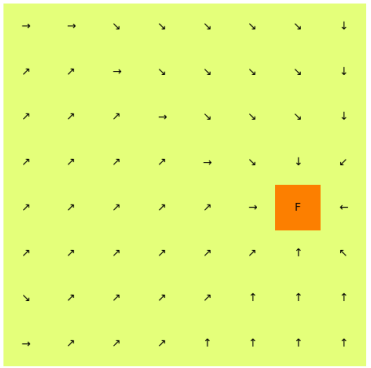
- In the left we can see how Q-learning differs from different learning rate.
  - **0.2** is a good trade-off between the new information learned from the current experience and the old Q-value
- In the right we can see how change *Policy iteration* with different type of piece
  - The algorithm always converge with the optimal policy based on the type of piece

Results

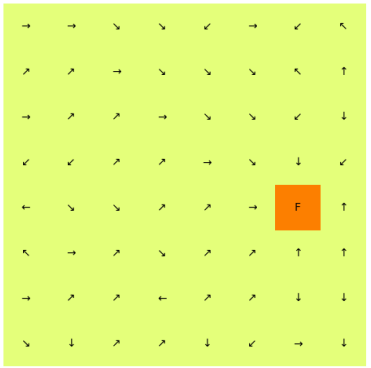
Optimal policy



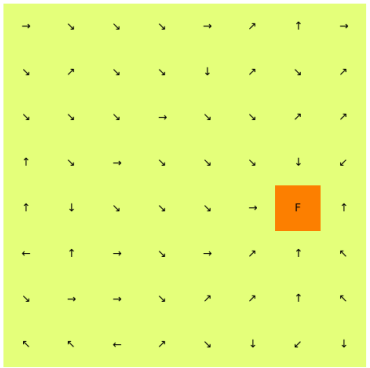
Policy Iteration



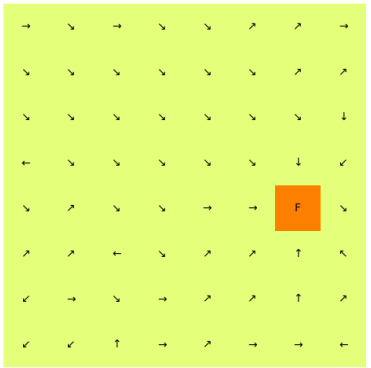
Value Iteration



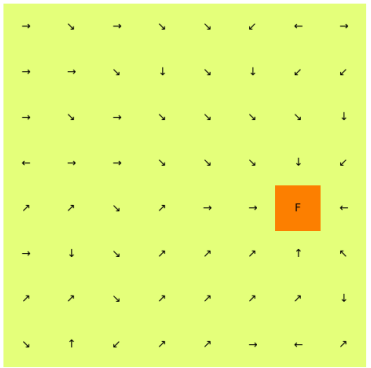
Q-Learning



SARSA



Exp SARSA



SARSA LAMBDA

Capture pieces

Introduction

Goal

Train a software agent to play a soft version of chess via Deep Q-Learning.

Game

As white player, capture valuable pieces of a random opponent in n moves, not for checkmate.

Why

It's a model-free RL project with total observability.
The game is discrete as it has only a finite number of n moves.
Since chess has state space complexity of 10^{46} [1], Q-network is ideal instead of Q-tables.

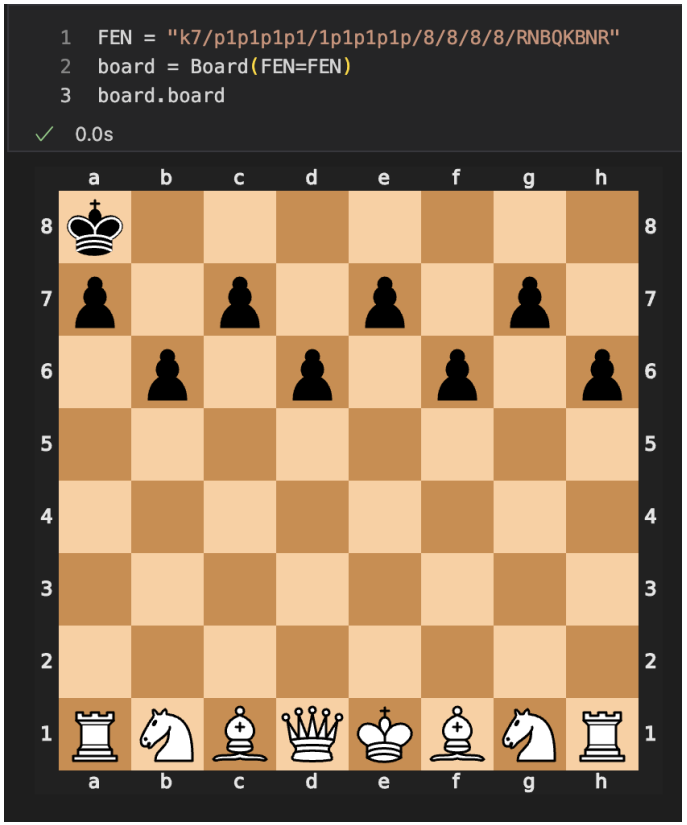
Environment

Based on the Board object from python-chess [2].

- init configuration set via FEN [3] (or default)
- n=25 maximum number of moves, after: reset env
- agent plays white
- black player returns random moves

```
1  FEN = "k7/p1p1p1p1/1p1p1p1p/8/8/8/8/RNBQKBNR"
2  board = Board(FEN=FEN)
3  board.board
```

✓ 0.0s



Environment

State

Our state is represent by an $(8x8x8)$ array:

- $(8x8)$ is the chess board dimension
- 8 are the dimensions of the pieces:
  - $0 \rightarrow$ pawns
  - $1 \rightarrow$ rooks
  - $2 \rightarrow$ knights
  - $3 \rightarrow$ bishops
  - $4 \rightarrow$ queens
  - $5 \rightarrow$ kings
  - $6 \rightarrow$ reciprocal of the current move number
  - $7 \rightarrow$ can-claim-draw
- white pieces value $= +1$
- black pieces value $= -1$

Rewards

A piece-centric reward system is used, where the agent receives points based on the type of piece it captures:

- +1 for pawns
- +3 for knights and bishops
- +5 for rooks
- +9 for queen

This is the most common assignment of point values for capturing pieces (known as Reinfeld material values): have no role in the rules of chess but serves only to plan strategic moves. Other more complex systems of valuation exists.

Agent and actions

The agent is no longer a single piece, it's the white chess player.
Each action is a tuple representing a potential move (from/to).
Its action space consist of $64x64 = 4096$ actions:

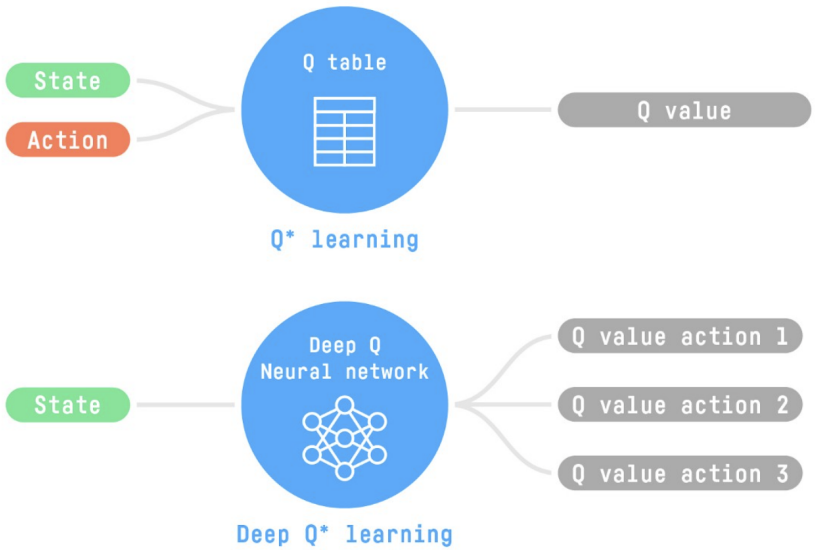
- $8x8 = 64$ places from where a piece can be picked up.
- $8x8 = 64$ places from where a piece can be dropped.

Of course, only certain actions are legal: we used the python-chess package to select legal moves.

Deep Q-learning

Intro

Q-Learning is an RL technique that trains Q-function to calculate the expected future reward for a state-action pair, helping the agent to learn the best actions.
In DQL the Q-function is approximated using a neural network due to the vast number of possible states in chess.



DQL

Algorithm

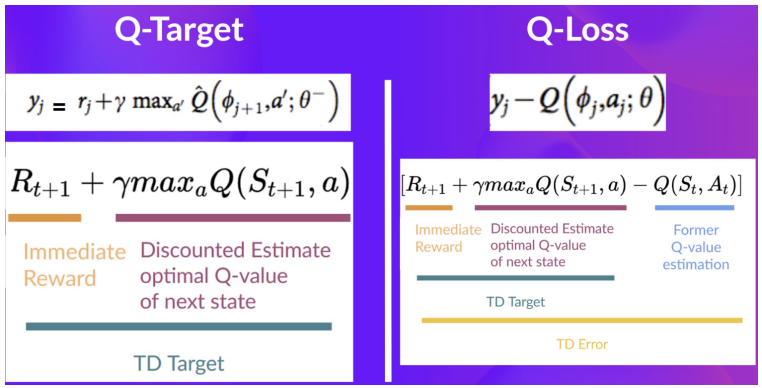
Instead of updating the Q-value of a state-action pair directly, in DQL loss function compares our Q-value prediction and the Q-target and uses SGD to update the weights of our network to approximate Q-values better.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma max_a Q(S_{t+1}, a) - Q(S_t, A_t)]$$

New Q-value estimation | Former Q-value estimation | Learning Rate | Immediate Reward | Discounted Estimate optimal Q-value of next state | Former Q-value estimation

TD Target

TD Error



Pseudocode

Algorithm

Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^{-} = \theta$
For episode = 1, M **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  For $t = 1$,T **do**
    With probability ε select a random action a_t
    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^{-}) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to the network parameters θ
    Every C steps reset $\hat{Q} = Q$
  End For
End For

Experience Replay

Fixed Q-Target

DQL has two phases:

- Sampling: we perform actions and store the observed experience tuples in a replay memory.
- Training: Select a small batch of tuples randomly and learn from this batch using SGD update step.

Experience-Replay

Algorithm

DQL training might suffer from instability, because of combining a non-linear Q-value function (NN) and bootstrapping (when we update targets with existing estimates and not an actual complete return).Thus we used Experience Replay. It has two functions:

- make more efficient use of the experiences during the training.
- avoid forgetting previous experiences and reduce the correlation between experiences.

Fixed-Q target

Algorithm

When we want to calculate the TD error (aka the loss), we calculate the difference between the TD target (Q-Target) and the current Q-value (estimation of Q).
But we don't have any idea of the real TD target. We need to estimate it. Using the Bellman equation, we saw that the TD target is just the reward of taking that action at that state plus the discounted highest Q value for the next state.
Consequently, there is a significant correlation between the TD target and the parameters we are changing.
Therefore, at every step of training, both our Q-values and the target values shift. We're getting closer to our target, but the target is also moving. It's like chasing a moving target! This can lead to significant oscillation in training.
Instead, what we see in the pseudo-code is that we:

- Use a separate network with fixed parameters for estimating the TD Target.
- Copy the parameters from our Deep Q-Network every C steps to update the target network.

Parameters

Algorithm

In our case:

- episodes are the games (usually 700)
- exploration rate:

$$\epsilon = \begin{cases} \max\left(0.05, \frac{1}{1+\frac{k}{250}}\right) & \text{if not greed} \\ 0 & \text{otherwise} \end{cases}$$

  with k game count
- update Q_{target} every c=10 games
- capacity of the replay memory is set to 1000 by default, but it can be customized

Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
For episode = 1, M **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  For t = 1,T **do**
    With probability ε select a random action a_t
    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to the network parameters θ
    Every C steps reset $\hat{Q} = Q$
  End For
End For

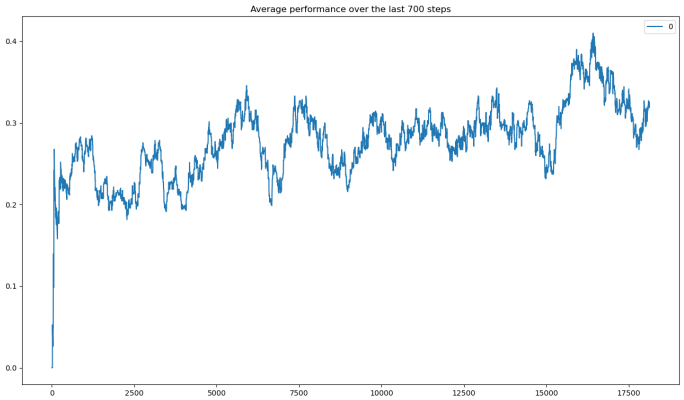
Q-network

The Q-network is usually either a linear regression or a (deep) neural network. The idea is similar to learning with Q-tables. We update our Q value in the direction of the discounted reward + the max successor state action value.

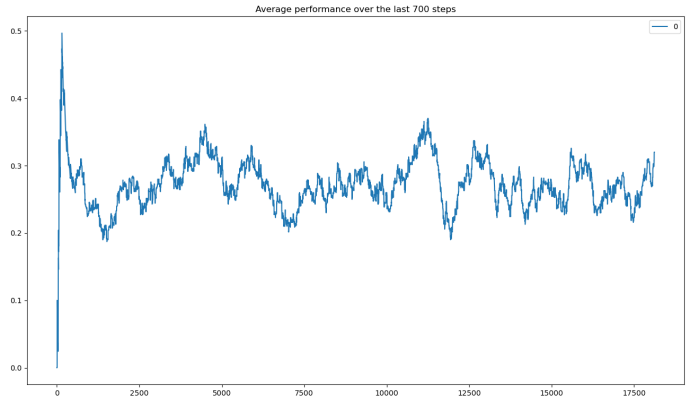
The input of the network is the state of the board ($8x8x8 = 512$) and the output is the predicted action value of each action ($64x64 = 4096$).
We used a MLP network with two hidden layers of dim 256 and two ReLu activation functions.
Network uses SGD and is optimized via Adam optimizer.
The Loss function is the Mean Squared Error (MSE), which measures the difference between the estimated and actual Q-values.

Results - Part 1

Results show the moving average (every 25 moves) for the rewards.

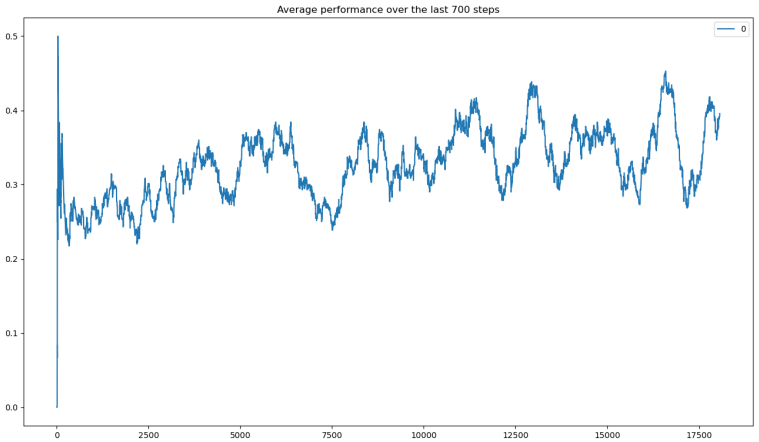


(a) $\gamma = 0.1$, $lr = 0.07$

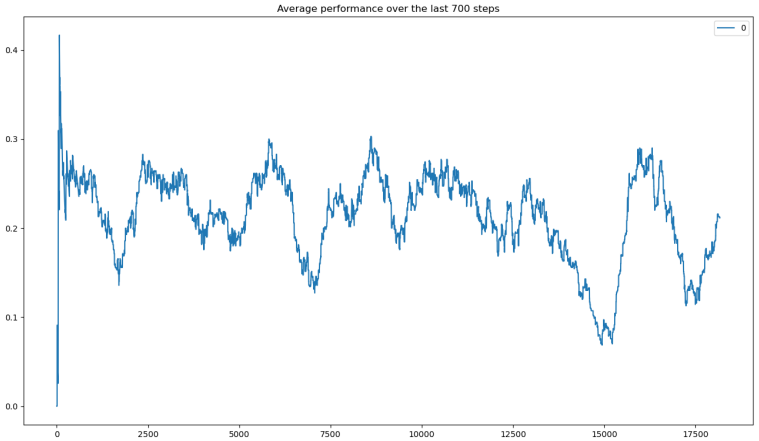


(b) $\gamma = 0.5$, $lr = 0.01$

Results - Part 1

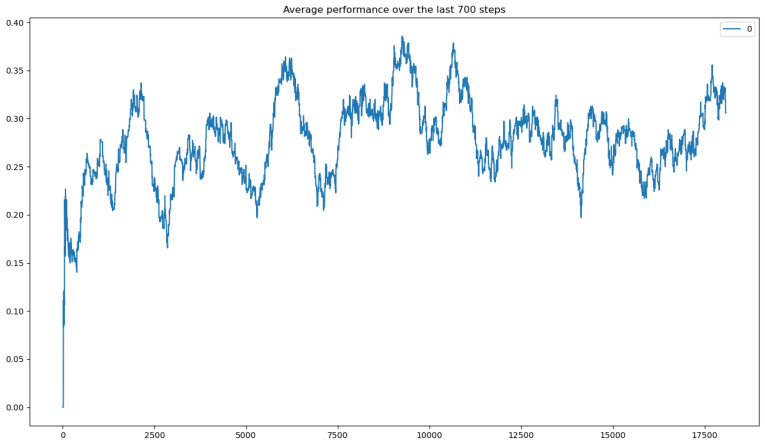


(a) $\gamma = 0.5$, $lr = 0.03$

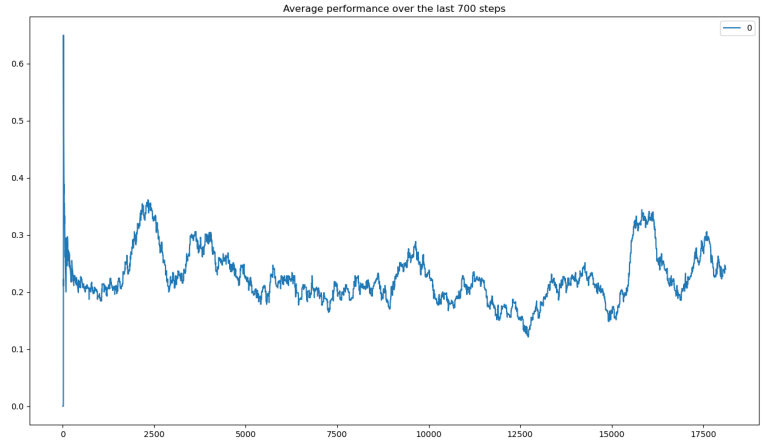


(b) $\gamma = 0.95$, $lr = 0.001$

Results - Part 1



(a) $\gamma = 0.95$, $lr = 0.01$



(b) $\gamma = 0.95$, $lr = 0.07$

Action-values

What has the network learned?

We expect that the action values for capturing black pieces is similar to the (Reinfeld) rewards we put in our environment. Of course the action values also depend on the risk of re-capture by black and the opportunity for consecutive capture.
The img in next slides checks the action values of capturing every black piece for every white piece.

Results - Part 2

Assume we are in move 10

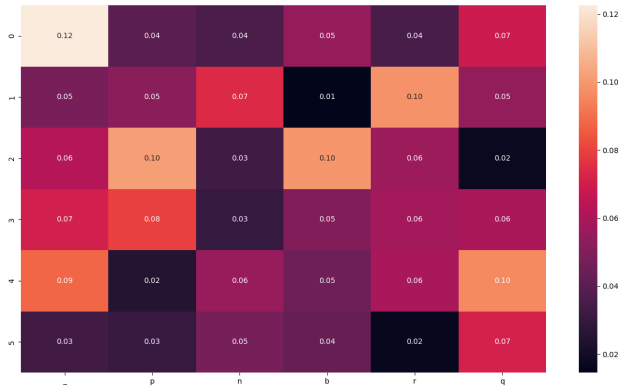


Figure: $\gamma = 0.1$, $lr = 0.07$

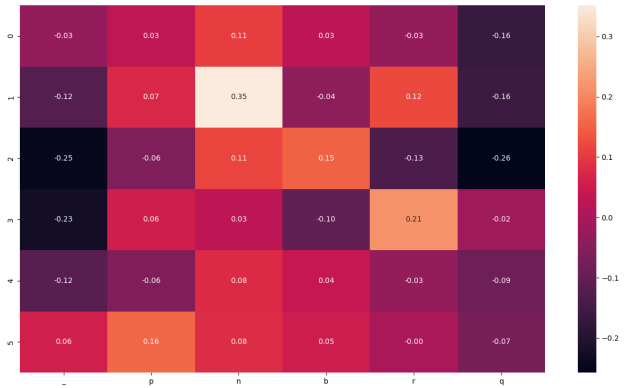


Figure: $\gamma = 0.5$, $lr = 0.01$

Results - Part 2

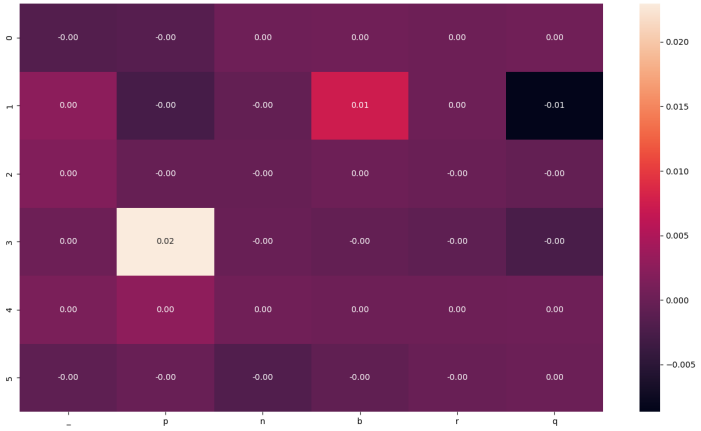


Figure: $\gamma = 0.5$, $lr = 0.03$

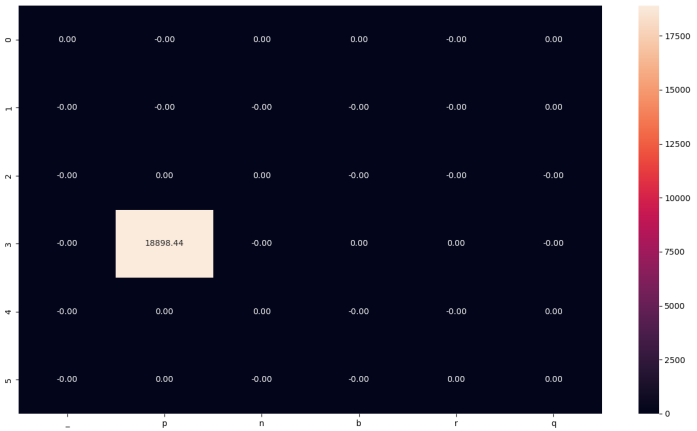


Figure: $\gamma = 0.95$, $lr = 0.001$

Results - Part 2

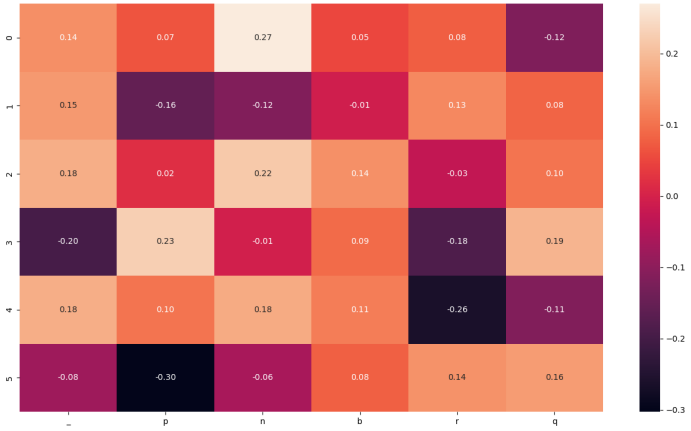


Figure: $\gamma = 0.95$, $lr = 0.01$

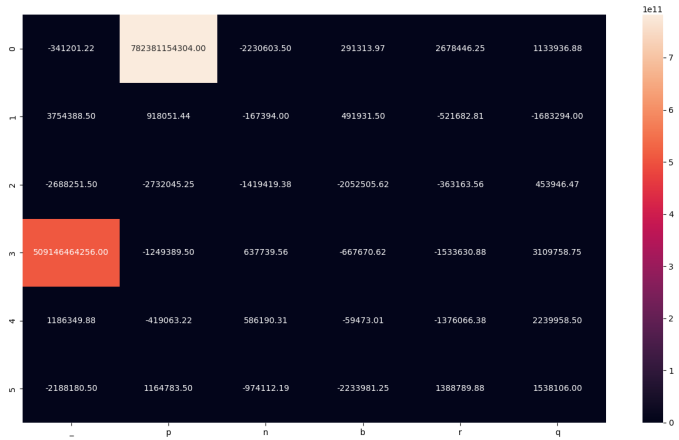


Figure: $\gamma = 0.95$, $lr = 0.07$

Final game video

Click here to open the Chess.com Analysis and play it

ReinForceMate: An Overview

ReinForceMate is a Python library buil on top of PyTorch that provides two environments within the domain of chess.

- `Capture Pieces`: This environment allows for a simpflified version of a chess game. The environment supports calculating valid moves, making a move, calculating rewards based on piece captures, and determining whether the game has ended.
- `Shortest Path`: This is a simplified environment where the chess board is treated as an 8x8 grid. The goal of the agent in this environment is to navigate to a specific terminal state from a starting state.

Library Structure

The library is organized into three main directories:

- `Agent`: Contains the files to implement agent capabilities such as capturing and moving.
- `Environment`: Includes the files to set up the environment for the agent including the game rules for capturing and moving.
- `Learn`: Houses the implementation of various reinforcement learning algorithms discussed earlier.

Agent: Capture

This class represents a Q-learning agent with a simple linear network.

- `init_optimizer`: Initializes the optimizer (Adam) for the neural network.
- `create_model`: Initializes a linear neural network model.
- `init_linear_networks`: Implements the Fixed-Q algorithm where it initialize the optimizer and creates a clone of the main model.
- `network_update`: Updates the Q-network using samples from a minibatch of experiences (experience replay). The network update is done by computing the Q-values and the target Q-values using the Bellman equation.
- `get_action_values`: Returns action values for a given state by forward propagating it through the model.

Environment: Capture Pieces

This class is used to simulate a simpfligame of chess with different methods to initialize the game board

- `step`: This method executes an action and updates the environment state. It returns a boolean indicating if the game has ended and the reward gained from the move.
- `get_random_action`: This method returns a random legal move from the current board state.
- `project_legal_moves`: This method projects the legal moves to the action space. It updates the action space with 1's at the indices corresponding to legal moves.
- `get_material_value`: This method calculates the total material value on the board using Reinfeld values.

Learn: Capture Chess

- `learn`: Runs the Q-learning algorithm for specified iterations and updates the target network every c games. On the final iteration, the agent plays a greedy game.
- `play_game`: Plays a single game using an epsilon-greedy strategy to select actions. The agent either explores (random action) or exploits (action with highest Q-network value). All legal moves with maximum action value are considered. The game state, action, reward, and new state are recorded in memory.
- `sample_memory`: Retrieves a mini-batch of experiences from the memory based on a probability distribution defined by the absolute Temporal Difference (TD) errors. The experiences with higher TD error have a higher probability of being chosen.
- `update_agent`: It uses a mini-batch of experiences from memory to update the Q-network and sets the sampling probabilities in memory using the absolute value of TD errors.

Agent: Piece

This class is a piace agent that uses different movement strategies based on the piece type.

- `apply_policy`: method is used to apply the current policy of the agent, given the current state and a probability for exploration. It returns the selected action for the state under the current policy.
- `compare_policies`: method compares the current policy with the previous one, and returns the sum of absolute differences between them.
- `init_action_space`: method initializes the action space for the piece based on its type.
- `get_action`: method returns the action for a given state according to the current policy.

Environment: Shortest Path

The class initialize a grid 8x8, with a starting state and a terminal state

- `step`: This method performs an action in the grid world, which results in moving to a new state and receiving a reward. It returns the reward for the performed action and a flag indicating whether the episode has ended or not. If the new state is outside the boundaries of the grid, it remains in the old state.
- `render`: This method updates the visual representation of the grid world. The current state is marked by '[S]' and the terminal state is marked by '[F]'.

Learn: Shortest Path

- `visualize_policy`: This function creates a visual representation of the policy. It maps each action to an arrow direction, replacing the terminal state with ”F”. It is used to better understand the learned policy.
- `visualize_action_function`: is used to print out the maximum expected total reward for each possible action in each state.

Future Improvements

In Shortest Path:

- Try also with MC methods

For Capture Pieces:

- use a different valuation system [AlphaZero].
- train model for $>$ 1000 games.
- use a double DQL (for overestimation of Q-values).
- schedule a decay for learning rate (e.g. exponential) in NN.
- change network with ConvNN.
- use TD(λ).
- train white player vs our trained model for smaller number of games.

References


Chinchalkar, Shirish
ICGA Journal 1996
An upper bound for the number of reachable positions


Python Chess. *Python Chess Documentation*.
`https://python-chess.readthedocs.io/en/latest/index.html`.


FEN. *FEN Wikipedia*. `https://en.wikipedia.org/wiki/Forsyth%E2%80%93Edwards_Notation`


Tomašev, Nenad, Paquet, et al.
ArXiv preprint 2009.04374 2020
Assessing game balance with AlphaZero: Exploring alternative rule sets in chess