Year 2021-2022

Reinforcement Learning

Master 2, Data Sciences

Article:

Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm. David Silver, Thomas Hubert, Julian Schrittwieser, et al. 2017

Authors

Lévi-Dan Azoulay, Ali Bellamine, Nathane Berrebi.

l'X ÉCOLE POLYTECHNIQUE

INSTITUT POLYTECHNIQUE DE PARIS

Understanding and re-implementing a general reinforcement learning algorithm for Chess and Shogi

Running title Understanding and implementing *AlphaZero*

Authors

Lévi-Dan Azoulay[1*], Ali Bellamine[1*], Nathane Berrebi[1*].

Affiliations

[1] Institut Polytechnique de Paris (École Polytechnique, Télécom, ENSTA, ENSAE), 5 Av. Le Chatelier 2ème étage, 91764 Palaiseau, France.

* **Contributed equally**

Word count in the text: 5798 ; **Word count in title**: 12

Number of Tables: 0 ; **Number of Figures**: 6 ;

Reference count:

Appendix: 1

Keywords: reinforcement learning, chess, go, shogi, deepmind, monte-carlo tree search, residual neural network, alphazero.

The objective of the paper is to report on the reinforcement learning algorithm called *AlphaZero* developed by Deep Mind (a Google owned cutting-edge artificial intelligence company). *AlphaZero* has been developed to virtually master any game without using any additional domain knowledge other than the rules of the game. The authors successfully applied their algorithm to the game of Chess and Shogi. They report the proof-of-concept that a general-purpose reinforcement learning algorithm can achieve superhuman performance across different challenging domains without human domain-knowledge.

When it comes to reinforcement learning applied to board games, two milestone achievements come to mind. First, the striking victory of Deep Blue (an IBM chess-playing supercomputer) over Gary Kasparov, the Russian chess grandmaster. Second, the competition between *AlphaGo* and the south-Korean Go world champion Lee Sedol. Although these episodes may seem similar, there is a huge difference that lies in the way that *DeepBlue* and *AlphaGo* truly operate. While *DeepBlue* executes an alpha-beta algorithm in parallel that seeks to decrease the number of nodes that are evaluated by the minimax algorithm in a search tree, *AlphaGo* relies on state-of-the-art methods of artificial intelligence, such as deep learning and cutting-edge reinforcement learning algorithms. *AlphaZero* further extends this artificial intelligence achievement by learning and training by self-play only, as we will see. Our goal in this work is to decipher the reinforcement learning methods that were used to develop such an algorithm. We also present *DeepChess*, an *AlphaZero*-like algorithm that we developed to put to test our understanding of *AlphaZero*.

As stated in the original paper[1], the *AlphaZero* algorithm is a more generic version of the *AlphaGo Zero* algorithm that was first introduced in the context of Go. It replaces the handcrafted knowledge and domain-specific augmentations used in traditional game-playing programs with deep neural networks and a *tabula rasa* reinforcement learning algorithm. The key points that make *AlphaZero* so special are the following:

- *AlphaZero* entirely learns using self-play. It does not use any professional human game plays, nor any other domain-knowledge than the rules of the game.

- *AlphaZero*'s basic ingredients are (1) a deep neural network and (2) a Monte-Carlo Tree-Search algorithm.
- *AlphaZero* uses only *one* neural network for *both* policy *and* value evaluation.

I – Understanding *AlphaZero*

A- Reinforcement Learning[2]

The purpose of Reinforcement Learning (RL), an area of Machine Learning, is to develop an algorithm (or an *agent*) that can learn how to take certain actions in a given *environment*. The environment tells the agent about its *state,* both *before* taking an action and *after* taking an action.

- **Policy**

What defines the learning agent's way of behaving at a given time is called the *policy*. Roughly speaking, a policy is a mapping from perceived states of the environment to actions to be taken when in those states. The policy is the core of a reinforcement learning agent. In general, policies may be stochastic, specifying probabilities for each action.

- **Reward**

On each time step, the environment sends to the reinforcement learning agent a single number called the *reward*. The reward signal thus defines what are the good and bad events for the agent.

Whereas the reward signal indicates what is good in an immediate sense, a *value function* specifies what is good in the long run. Roughly speaking, the *value* of a state is the total amount of reward an agent can expect to accumulate over the future, starting from that state. To make a human analogy, rewards are somewhat like pleasure (if high) and pain (if low), whereas values correspond to a more refined and farsighted judgment of how pleased or displeased we are that our environment is in a particular state.

Nevertheless, without rewards there could be no values, and the only purpose of estimating values is to achieve more reward. But recall that it is values with which we are most concerned when making and evaluating decisions. *The policy relies on the value function.* We seek actions that bring about states of highest value, not highest reward, because these actions obtain the greatest amount of reward for us over the long run.

Unfortunately, it is much harder to determine values than it is to determine rewards. Rewards are basically given directly by the environment, but values must be estimated and re-estimated from the sequences of observations an agent makes over its entire lifetime.

- **Value**

The value function of a state s under a policy π , denoted v⇡(s), is the expected return when starting in s and following π thereafter. For MDPs, we can define v_π formally by:

$$v_\pi(s) \doteq \mathbb{E}_\pi[G_t \mid S_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,\middle|\, S_t = s\right], \text{ for all } s \in \mathcal{S},$$

where E$_\pi$[·] denotes the expected value of a random variable given that the agent follows policy π, and t is any time step. Note that the value of the terminal state, if any, is always zero. We call the function v⇡ the state-value function for policy π.

Similarly, we define the value of taking action a in state s under a policy π, denoted $q_\pi(s,a)$, as the expected return starting from s, taking the action a, and thereafter following policy π:

We call q_π the action-value function for policy π

$$q_\pi(s,a) \doteq \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,\middle|\, S_t = s, A_t = a\right].$$

- **Environment model**

To grasp the environment's behavior, on can need a *model*, which is something that mimics the behavior of the environment, or more generally, that allows inferences to be made about how the environment will behave. For example, given a state and action, the model might predict the resultant next state and next reward. Models are used for *planning*, by which we mean any way of deciding on a course of action by considering possible future situations before they are actually experienced. Methods for solving reinforcement learning problems that use models and planning are called model-based methods, as opposed to simpler model-free methods that are explicitly trial-and-error learners—viewed as almost the opposite of planning.

B- High level view of AlphaZero[3]

Although the essence of *AlphaZero* is to be applicable to any game, in order to be both thorough and clear, we will mainly focus our attention to the game of chess.

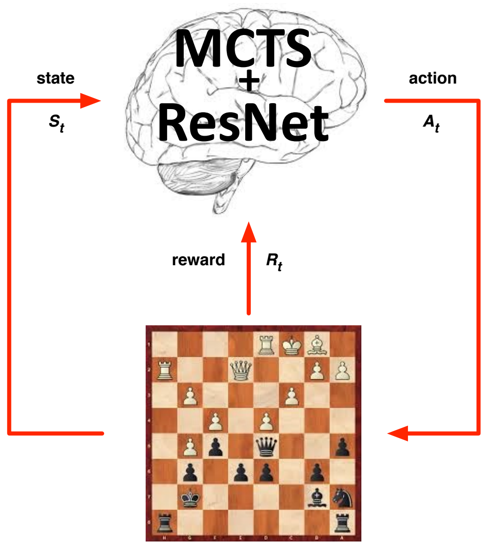
In *AlphaZero*, *states* are represented by the board positions, the *environment* is modelized by the rules of the game, which includes the legal moves. An *action* is a move, picked among all legal moves (finite ensemble). Actions depends on states. The *policy* is the way the agent (player) picks its moves. The policy reflects the agent's strategy to win the game. The *agent* is the player.

It is a *model-based RL algorithm* for it relies on a model of the environment (rules of the game). This allows the agent to learn from simulated games rather than from actual played games.

The *planning* part is performed using a *Monte-Carlo Tree Search (MCTS)*

To *modelize* the policy, *AlphaZero* uses a *deep neural network* (p, v) = f_θ (s) with parameters θ to perform search value and policy predictions

Figure 1. High-level view of *AlphaZero*

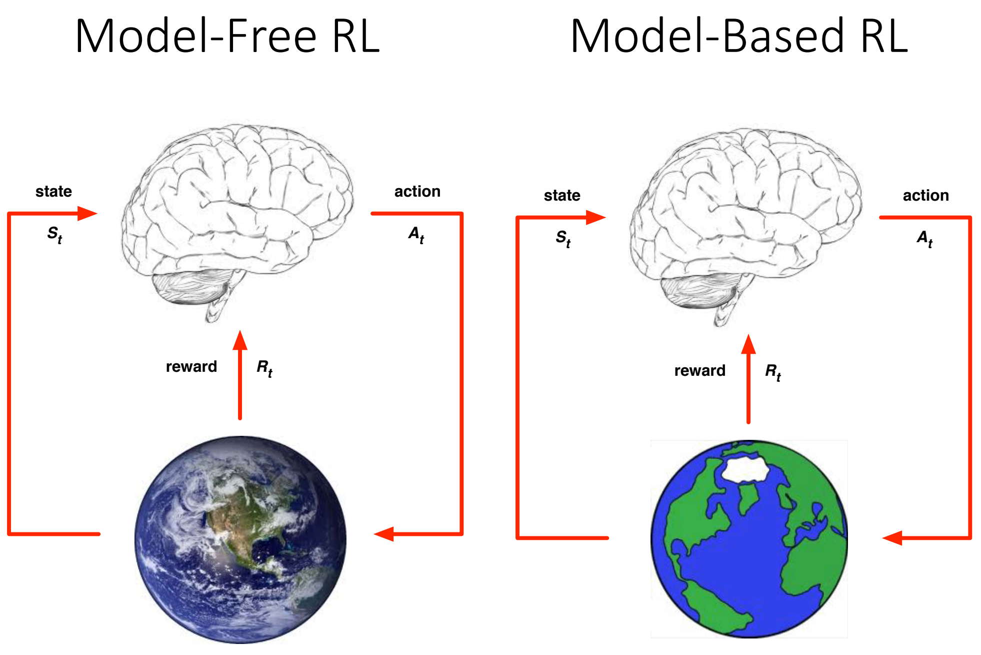


The idea behind *AlphaZero* is to entirely learn using self-play. The objective was to implement a superachieving *tabula rasa* reinforcement learning algorithm. The authors clearly state that the only domain knowledge that was given to *AlphaZero* are the rules of the game.

In order for the algorithm to generalize well, the model has to be as simplest as it can be. **Therefore, the model is nothing but the rules of the game and the game board. The agent's policy is given by the deep neural network, which is trained using the Monte-Carlo tree search algorithm.**

Monte-Carlo Tree Search, which is used in *AlphaZero* is a model-based reinforcement learning algorithm. Model-Based RL consists in learning from a model directly rather than from experience. In this setting, the model is what describes the agent's understanding of the environment (Figure 2). The model tells us about (1) how states transition to other states (transition dynamics) and (2) how states lead to rewards (reward function). **The agent is interacting with its own model to learn.**

Figure 2. Model-Free versus Model-Based RL

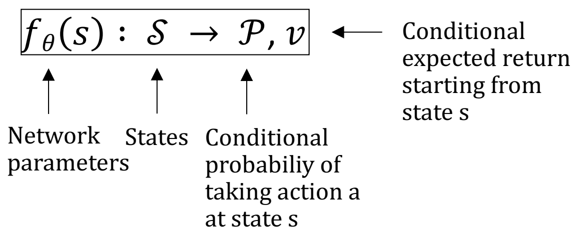


The advantage of Model-Based RL is that a model is a better representation for the environment than a "simple" value function.

C - AlphaZero’s deep neural network and policy modelisation

***AlphaZero* uses a deep neural network (p, v) = f_θ (s) with parameters θ to perform search value and policy predictions** (Figure 2). The neural network takes the board position s as an input and outputs a vector of move probabilities p with components $p_a = Pr(a|s)$ for each action a, and a scalar value v estimating the expected outcome z from position s, $v \approx E[z|s]$. *AlphaZero* learns these move probabilities and value estimates entirely from self- play.

Figure 3. *AlphaZero*’s deep neural network


$f_\theta(s) : \mathcal{S} \rightarrow \mathcal{P}, v$
Conditional expected return starting from state s
Network parameters
States
Conditional probabiliy of taking action a at state s

We can use a supervised learning framework (find θ such that we minimize a loss $\mathcal{L}$ that measures the discrepancy between $f_\theta(x)$ and y (namely, $p_a = Pr(a|s)$ and $v \approx E[z|s]$). The parameters θ of the deep neural network in *AlphaZero* are trained by using **records** of previous self-played games, starting from randomly initialized parameters θ.

- ***AlphaZero*’s Neural network’s input and output**

One of the issues that arise when trying to model a chess move, with a machine learning based solution, is how to ensure to get a fixed-size input and output. The authors solved this problem by taking into account any possible board configuration and any possible move.

The input is a matrix representation of the chessboard, it can be perceived as an 8x8 image. They one-hot encoded the player-piece information. This gives, for each time a (8x8x12) tensor. Additional needed informations are not encoded inside the board, this is for example the number of played move, the king and queen castling situation for the current player. This L informations could be represented as a vector, but the author stated that they stored it inside an 8x8 matrix. We supposed that they reshaped them to make them fill inside the global input tensor.

Suppose there are T time steps, set M = 12 an N = 8, this gives us an input tensor of size (N x N x (MT + L)).

The network output is composed of two elements:
- The moves probability distribution over all possible moves.
- The value estimation.

The moves are encoded through a tensor of (8x8xN) size, with N the total number of possible moves. Each (8x8) matrix represents the probability of player one for the N'th move for each piece of the current board.
It is a well-known fact from chess players that there exist 73 different moves. These moves correspond to the "queen moves", which encode all the straight moves in all the possible directions (N, NE, E, SE, S, SW, W and NW) and all the possible distances (from 1 to 8), the knights moves in the 8 possible directions and under-promotions which are defined as a promotion under the queen piece (knight, bishop, rook).
We wanted to say a word about the castling, which is a little subtle. This move is actually only described by the king's move which is actually legal only in case of a castling.

This moves encoding does not take into account the legality of the moves. The final moves distribution is given by filtering the moves over the legal ones.

The value estimation is simply a scalar.

- ***AlphaZero*'s Neural network's architecture**

AlphaZero's neural network is mainly composed of convolutional layers.
The authors distinguished the "head" and the "body" part of the network.

The body encodes the representation of the board by an intermediate tensor representation of size (8x8x256). This is provided by 19 residuals blocks composed of two 256-filters convolution layers of kernel size (3,3) and stride (1,1) followed by a batch normalization and a ReLU activation function. Between each residual block, a skip-connection is performed.

The head is composed of two independent components, the head for moves estimation and the one for value evaluation.

The head, which estimates the move to play, is composed by a stack of convolutional networks with batch normalization and ReLU activations. It ends with a 73 filters convolutional network followed by a Softmax activation to encode the probability distribution of the move to play.
The value head is also composed of the stack of convolutional networks with batch normalization and ReLU activation and it ends by a flattening followed by a linear layer and a Tanh activation function.

- **The neural network's loss**

The neural network parameters θ are **updated** so as to minimize the error between the *predicted* outcome v and the *true* game outcome z, and to maximize the similarity of the policy vector p to the search probabilities π, which represent the set of possible actions to take at a given state. Specifically, the parameters θ are adjusted by gradient descent on a loss function $\mathcal{L}$ that sums over mean-squared error and cross-entropy losses respectively:

$$\mathcal{L} = (z-v)^2 - \boldsymbol{\pi}^{T} \log \mathbf{p} + c||\theta||^2$$

where c is a parameter controlling the level of L_2 weight regularization.

- **Policy modelization: the link between the neural network and the policy**

The neural network allows us to generate a move distribution according to its trained parameters. This is the first part of the policy modelization, which will be further refined by game simulations performed by the MCTS. It's the addition of the neural networks output and the MCTS simulation that constitutes the actual algorithm policy.

But what exactly do we mean by Monte-Carlo Tree Search and self-play? How do we properly generate a training set?

D- Monte-Carlo Tree Search (MCTS) and self-play

Chess has around 10^{48} legal board positions. This overwhelming number of combinations is the reason why brute force, classic search tree and minimax algorithms do not scale well. This is called the problem of branching factor.

The answer to address this problem is the **Monte-Carlo tree search** algorithm. It only explores **some** paths. It restricts the path search to the most interesting states. **It simulates the games** using self-play.

- **Monte-Carlo Tree Search (MCTS)**

MTCS is state of the art method for the planning part of our RL goal and the one that is used in *AlphaZero*. Sutton and Barto qualify it as "a recent and strikingly successful example of decision- time planning". MCTS relies on forward search. Basically, MCTS builds a statistics tree that partially maps onto the entire game tree. Such tree further guides the simulation to look *only or at least mostly* at the interesting nodes in the game tree. The core idea of MCTS is to successively focus multiple simulations starting at the current state by **extending the initial portions of trajectories that have received high evaluations from earlier simulations.**

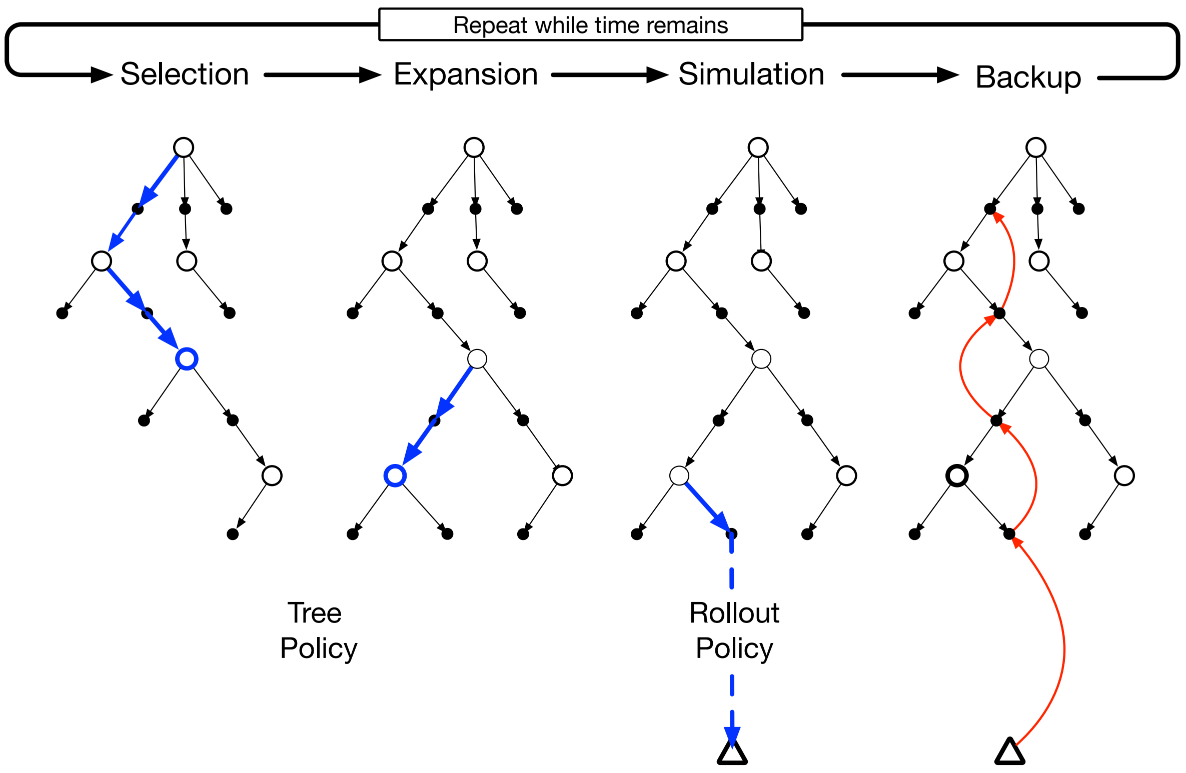
Each search consists of a series of simulated games of self-play that traverse a tree from root s_{root} to leaf. **Each simulation proceeds by selecting in each state s an action a according to its default or tree policy, balanced by a reward, a visit count and an eventual additional noise.**

Each simulation is run a certain number of time. The simulation is performed until it reaches a leaf node. Reaching a leaf node is what ends the simulation and triggers the **backpropagation** (Figure 5) which updates the reward that is expected for each state-action pair.

Starting with an empty tree, MCTS iteratively builds up a portion of the game tree by running a simulation of the game. Each iteration consists of three stages:

1. **Selection**. Starting at the root node, an action is chosen according to the tree policy
2. **Expansion**. We keep playing and simulating the next moves, using the tree policy.
3. **Simulation**. From the expanded chosen node, a simulation of a complete episode is run with actions selected by the default or rollout policy. At the end, we obtain the result of a Monte Carlo trial.
4. **Backup.** The return generated by the simulated episode is backed up to update, or to initialize, the action values attached to the edges of the tree traversed by the tree policy in this iteration of MCTS. Figure 4. illustrates this by showing a backup from the terminal state of the simulated trajectory *directly* to the state–action node in the tree where the rollout policy began.

Figure 4. Monte-Carlo Tree Search



When the environment transitions to a new state, MCTS executes as many iterations as possible before an action needs to be selected, incrementally building a tree whose root node represents the current state. Each iteration consists of the four operations: Selection, Expansion (though possibly skipped on some iterations), Simulation, and Backup. *Adapted from Sutton and Barto (2017) and Chaslot, Bakkes, Szita, and Spronck (2008).*

- **MCTS in *AlphaZero***

In *AlphaZero,* there is no rollout policy. The tree policy is given by the neural network. The move selection is performed by an UCB-like (upper-confidence bound) computation with additional Dirichlet noise to the neural network move distribution prediction.

The **tree policy** is given by the neural network. ***AlphaZero* runs every MCTS with a tree policy. Each simulation proceeds by selecting in each state s a move a with low visit count, high move probability and high value (averaged over the leaf states of simulations that selected a from s) according to the current neural network f_θ.** Many simulated trajectories can be generated in a short period of time. Acording to the result of the simulation, the respective reward estimation of each state-action pair encountered is updated. Each simulation benefits from the informations learned through the previous backpropagated simulations. **In *AlphaZero*, 800 simulations are performed and the actual chosen move by the agent is selected according to the new estimated reward given by the MCTS.** This is what we call the actual policy of our algorithm (that the agent follows).

- **Upper-confidence bound tree (UCT)**

At each node, an action is selected, among legal moves, as follows:

$$a_t \;=\; \arg\max_a \big(Q(s_t,a) \,+\, U(s_t,a)\big)$$

with *U(s,a*) defined as follows (using a variant of the PUCT algorithm)*:*

$$U(s,a) \;=\; C(s)P(s,a)\sqrt{N(s)}/(1 \,+\, N(s,a))$$

$$C(s) = \log\left((1 + N(s) + c_{\text{base}})/c_{\text{base}}\right) + c_{\text{init}}\,.$$

where *N(s)* the parent visit count, c_{init} and c_{base} are predefined constants given by the authors and *C(s)* the exploration rate.

C(s) and the Dirichelet noise are added to force exploration. A Dirichelet noise is added to the prior probability *P(s,a)* given by the neural network, and *C* is an exploration parameter that grows slowly during search and translates mathematically the exploitation/exploration tradeoff. Indeed, as the neural network learns, and allows the MCTS to search through narrower paths, we can allow to increase C to further foster the exploration.

The UCB provides the upper-bound of the confidence interval of the value function and the Dir. Noise is used to force exploration.

As the search continues, each parent nodes *Q(s,a)* gets updated. After running a simulation, the parents nodes *Q(s,a)* are updated backwards. *Q(s,a)* are the ways to encode compactly the information that the MCTS has gathered by playing.

- **Self-play and dataset generation**

After each game, the MCTS provide us, for each state, a distribution of action-reward pair from which is derived a policy by a softmax.

Games are played by selecting moves for both players by MCTS, $a_t \sim \pi_t$: this is what is called *self-play.* **Playing by simulating games** is what provides our dataset formed by the state-action pair and their value (estimated after completion of the game)

$$\begin{aligned} S_1, A_1 &\rightarrow R_2, S_2 \\ S_2, A_2 &\rightarrow R_3, S_3 \\ &\vdots \\ S_{T-1}, A_{T-1} &\rightarrow R_T, S_T \end{aligned}$$

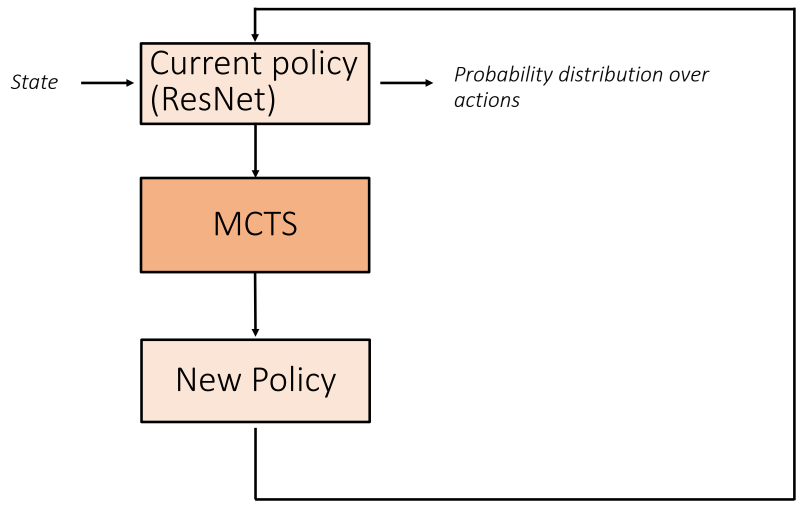
At the end of the game, the terminal position s_T gives a score z according to the rules of the game: −1 for a loss, 0 for a draw, and +1 for a win.

- **Link between the neural network and the MCTS**

We don’t *directly* train our network to make “good” moves. Instead, **we train it to mimic the behavior of the Monte-Carlo Tree Search**. As we play, the policy network suggests moves to Monte Carlo Tree Search. MCTS uses these suggestions (or priors) to “wisely” explore the game tree and returns a better set of probabilities for a given state. We record the state and the probabilities produced by the MCTS.

We use these self-played games records as a dataset to train our policy network. We ask our policy network what prior probabilities it would recommend for a given state. We then train our network, minimizing our loss function to try and match the policy network's priors to the probabilities recommended by the MCTS.

Figure 5. Search-based policy iteration



E- *AlphaZero*'s policy

After training, we don't only use the neural network output to chose the best action.

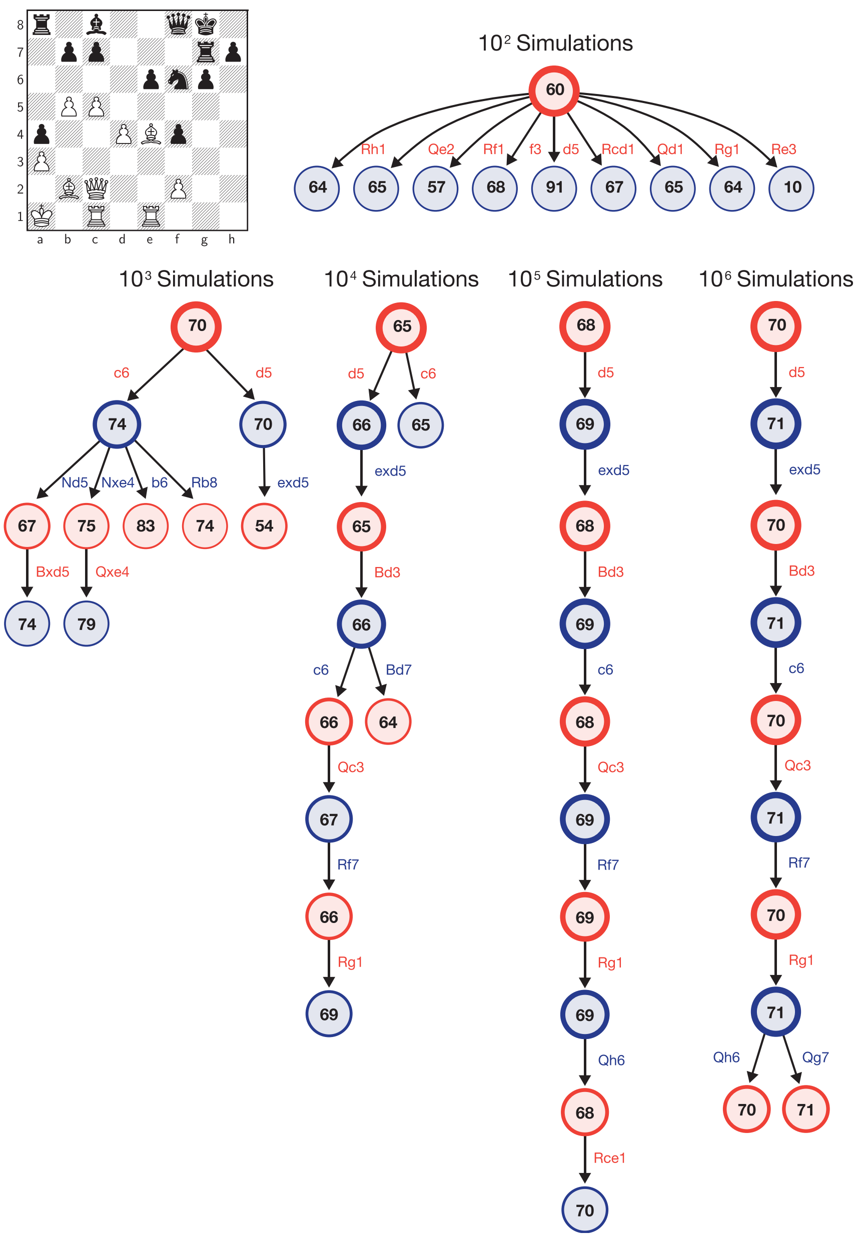
We keep running an MCTS, based on the neural network output.

<u>During an actual game, *AlphaZero* would:</u>

- Run simulations of self-play using MCTS with a tree policy (i.e using the trained neural network) (Figure 6)
- Backup the results of each simulation to the reward table, as in Figure 6.
- Pick the action according to the best updated reward (greedy policy) given by the last simulation

A key feature of *AlphaZero* in an actual game is to <u>use its MCTS</u> to <u>further refine the neural network output</u> in a way that the <u>final action to take at a given state s_t</u> is the one given by <u>*simulations of self-play starting at s_t using an already trained the neural network*</u>.

Figure 6. *AlphaZero's* search procedure during evaluation



F- Proof-of-concept

AlphaZero algorithm was applied to play chess, shogi, and also Go. Unless otherwise specified, the same algorithm settings, network architecture, and hyper-parameters were used for all three games. The authors managed to provide a proof-of-concept that a general reinforcement learning algorithm can achieve superior results within a few hours, searching a thousand times fewer positions, given no prior domain knowledge (other than the game's rules)

II– Re-implementing *AlphaZero*

We tried to re-implement the *AlphaZero* algorithm in order to play chess.
We are aware that the lack of computational resources is barrier to get an effective algorithm. This is the reason why we tried to develop DeepChess so as to provide an experimental and personal *AlphaZero*-like algorithm that can learn to play chess from scratch (*tabula rasa*), rather that to actually develop a truly powerful and effective algorithm.

The project is named DeepChess and its source-code had been made available publicly on GitHub under the MIT License. One may consult it at the following adress:
https://github.com/alibell/deepChess

The project is implemented in Python and is composed of multiple components:
- A python chess engine
- A neural-network modelisation of the policy
- An interface to play according to neural-network or Stockfish policy
- A Monte Carlo Tree Search implementation
- Self-play script

DeepChess is apparented to a library where each class does a particular mission. Each class is handled by a python script.

We implemented our own chessboard, but eventually used the quicker one called ShallowBlue.

A. Python chess engine

We decided to develop our own python chess engine class.
The motivation behind that is to allow to generate, in an suitable way, both the input (board representation) and move probability distribution for the neural network.
We represented the chess board with a *numpy* matrix of dimension (8,8). Each empty position is represented by a 0, each piece is represented by a an 8-bit integer in which the first number represents the piece (1- for pawn, 2- for rook, 3- for knight, 4- for

bishop, 5- for queen and 6- for king) and the second number the player (-0 for player 0 and -1 for player 1).

The chess board object can proceed to:

- Board evaluation (detect a win, lose or draw)
- Next legal move generation
- Generation of a Neural Network input (as described below)
- Generation of the legal moves list in the Neural Network output representation (as described below)
- Generation of the representation of the current board needed to interact with other chess engines
- Self-play and script evaluation

One of the drawbacks of our implementation lies in the legal move generation.
Indeed, the chess legal move computation is a non-trival issue.
It depends on many variables, such as the tower's or king's moves in the case of castling, the opponent's pawn move for « *en passant* » pawn move as well as the check identification, which requires to compute 1 to 2 legal moves in advance.

Even if the computation time of the legal moves is around 50 ms, this revealed to be too important to permit an efficient MCTS implementation.
Moreover, the lack of computational efficiency of Python contribute to the lack of computational speed of our chess board class.

In order to accelerate the next legal move computation, we created a faster version of our class which is connected to an efficient, programmed in C, chess engine, which is call *ShallowBlue* and referenced in the git repository.

B. Neural network representation of the chess engine

The neural network has been developed in Python using the PyTorch library.
Our neural network is mainly composed of convolutional layers (Appendix 1).

Its input is a board representation of each piece of each player for the eight last states concatenated with a representation of key informations about the current chess play : current player to play, castling state in the king and queenside, number of moves played, number of move without evolutions of the game, number of repetition of the same configuration of the board …
It all comes back to an **input tensor of size (97,8,8).**

The neural network architecture is similar as the one described in the supplementary material describing *AlphaZero*.
We can distinguish *a body* and *two heads*: value function and policy head.

The body is composed of the stack of 19 residual blocks. An individual residual block is successively composed of two convolution with kernel-size (3,3) and 256 filters, a batch normalization and a ReLU activation function, followed by a skip connection.

As a result, the neural network contains 23,419,973 millions parameters.

C. Players or interface to play according to neural-network or Stockfish policy

In order to interact with the chess board, we developed a python class which select move according to a given policy.
We called this class *Player*.

In order to process to some tests and evaluate the performance of the algorithm, we developed different kind of players :
- A Neural Network player: it selects the next moves according to neural network policy
- A k-Stockfish player: it selects the next moves according to a mix of random and stockfish policy

- **A few words about k-StockFish player**

We wanted to challenge our algorithm with Stockfish. We hypothesized that it won't be able to beat Stockfish (as we do not dispose of 6000 TPUs, it seemed for us to be a realistic hypothesis). Therefore, we created a weaker StockFish player called the k-Stockfish player. The k-Stockfish player selects at each turn its move according to either Stockfish's algorithm (with a probability k) or according to a random distribution (with a probability 1-k)

- **The link between the Monte Carlo Tree Search and the player**

As seen in the first part of this document, our actual algorithm policy consists in picking the move which maximize the estimation of the reward according to the MCTS.

One could argue that the MCTS could be seen as a form of Player. However, the MCTS is also a function of a player (which represents the tree policy). We therefore decided not to implement it as a player because of issues with recursive module import in Python.

D. Monte Carlo Tree Search

The implementation of the Monte Carlo Tree Search was done accordingly with the description provided by the authors. We also used the provided a Python pseudo which describe the functioning of their MCTS architecture.

We developed it as a class which can be instanced with a defined Tree Policy (in our case the Neural Network) and some tuning parameters.

Our MCTS class is able to give the next move according to a given number of simulations. During these simulations, it keeps in its internal memory a dictionary of the state-action pairs.
We choose to encode the state with the Fen string of the chess board and the action as a hash of the action.

At the end of the next-move selection, we are able to export the record of each encountered states with its action-reward distribution. This record is saved in a pickle file in a folder dedicated to the neural network training.

E. Self-play and neural-network training

We provided two scripts :

- selfPlay.py
- trainNN.py

We run them concurrently, the selfPlay script generates chess plays with the chess board engine, the neural network with MCTS plays again itself. It also plans to try to train the neural network to play against somes k-stockFish player.
During selfPlay, the generated record of the games are saved in the dedicated folder. The trainNN script recursively explores this folder and proceeds to a gradient descent on the neural networks parameters on the provided data. We fixed a limit in the number of time the same record is send to the neural network. The idea behind this is to always send the most recent records, which should represent the best neural network player.

To wrap up this discussion, let us recap the training steps for DeepChess:

- Player 0 considers the CNN's probability table (CNN's output)
- It simulates 800 games, using an MCTS framework. MCTS searches are run with a CNN policy combined with an UCB rule. The simulation runs only over the most promising moves predicted by the CNN.
- It backpropagates to each simulation tree its results.
- After 800 simulations, it updates the probability of each state-action pair.
- The player 0 picks the best one.
- The process goes on until we reach a lose, draw or win state.
- During this training, we stock every state-action pair of the game.
- We continuously feed these to the CNN (which improves over time). As it improves, the MCTS simulations also improve.

Appendix 1. DeepChess' neural network architecture.

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1            [2, 256, 8, 8]         223,488
       BatchNorm2d-2            [2, 256, 8, 8]             512
              ReLU-3            [2, 256, 8, 8]               0
            Conv2d-4            [2, 256, 8, 8]         589,824
       BatchNorm2d-5            [2, 256, 8, 8]             512
              ReLU-6            [2, 256, 8, 8]               0
            Conv2d-7            [2, 256, 8, 8]         589,824
       BatchNorm2d-8            [2, 256, 8, 8]             512
            Conv2d-9            [2, 256, 8, 8]         589,824
      BatchNorm2d-10            [2, 256, 8, 8]             512
             ReLU-11            [2, 256, 8, 8]               0
           Conv2d-12            [2, 256, 8, 8]         589,824
      BatchNorm2d-13            [2, 256, 8, 8]             512
           Conv2d-14            [2, 256, 8, 8]         589,824
      BatchNorm2d-15            [2, 256, 8, 8]             512
             ReLU-16            [2, 256, 8, 8]               0
           Conv2d-17            [2, 256, 8, 8]         589,824
      BatchNorm2d-18            [2, 256, 8, 8]             512
           Conv2d-19            [2, 256, 8, 8]         589,824
      BatchNorm2d-20            [2, 256, 8, 8]             512
             ReLU-21            [2, 256, 8, 8]               0
           Conv2d-22            [2, 256, 8, 8]         589,824
      BatchNorm2d-23            [2, 256, 8, 8]             512
           Conv2d-24            [2, 256, 8, 8]         589,824
      BatchNorm2d-25            [2, 256, 8, 8]             512
             ReLU-26            [2, 256, 8, 8]               0
           Conv2d-27            [2, 256, 8, 8]         589,824
      BatchNorm2d-28            [2, 256, 8, 8]             512
           Conv2d-29            [2, 256, 8, 8]         589,824
      BatchNorm2d-30            [2, 256, 8, 8]             512
             ReLU-31            [2, 256, 8, 8]               0
           Conv2d-32            [2, 256, 8, 8]         589,824
      BatchNorm2d-33            [2, 256, 8, 8]             512
           Conv2d-34            [2, 256, 8, 8]         589,824
      BatchNorm2d-35            [2, 256, 8, 8]             512
             ReLU-36            [2, 256, 8, 8]               0
           Conv2d-37            [2, 256, 8, 8]         589,824
      BatchNorm2d-38            [2, 256, 8, 8]             512
           Conv2d-39            [2, 256, 8, 8]         589,824
      BatchNorm2d-40            [2, 256, 8, 8]             512
             ReLU-41            [2, 256, 8, 8]               0
           Conv2d-42            [2, 256, 8, 8]         589,824
      BatchNorm2d-43            [2, 256, 8, 8]             512
           Conv2d-44            [2, 256, 8, 8]         589,824
      BatchNorm2d-45            [2, 256, 8, 8]             512
             ReLU-46            [2, 256, 8, 8]               0
           Conv2d-47            [2, 256, 8, 8]         589,824
      BatchNorm2d-48            [2, 256, 8, 8]             512
           Conv2d-49            [2, 256, 8, 8]         589,824
      BatchNorm2d-50            [2, 256, 8, 8]             512
             ReLU-51            [2, 256, 8, 8]               0
           Conv2d-52            [2, 256, 8, 8]         589,824
      BatchNorm2d-53            [2, 256, 8, 8]             512
           Conv2d-54            [2, 256, 8, 8]         589,824
      BatchNorm2d-55            [2, 256, 8, 8]             512
             ReLU-56            [2, 256, 8, 8]               0
           Conv2d-57            [2, 256, 8, 8]         589,824
      BatchNorm2d-58            [2, 256, 8, 8]             512
           Conv2d-59            [2, 256, 8, 8]         589,824
      BatchNorm2d-60            [2, 256, 8, 8]             512
             ReLU-61            [2, 256, 8, 8]               0
           Conv2d-62            [2, 256, 8, 8]         589,824
      BatchNorm2d-63            [2, 256, 8, 8]             512
           Conv2d-64            [2, 256, 8, 8]         589,824
      BatchNorm2d-65            [2, 256, 8, 8]             512
             ReLU-66            [2, 256, 8, 8]               0
           Conv2d-67            [2, 256, 8, 8]         589,824
      BatchNorm2d-68            [2, 256, 8, 8]             512
           Conv2d-69            [2, 256, 8, 8]         589,824
      BatchNorm2d-70            [2, 256, 8, 8]             512
             ReLU-71            [2, 256, 8, 8]               0
           Conv2d-72            [2, 256, 8, 8]         589,824
      BatchNorm2d-73            [2, 256, 8, 8]             512
           Conv2d-74            [2, 256, 8, 8]         589,824
      BatchNorm2d-75            [2, 256, 8, 8]             512
             ReLU-76            [2, 256, 8, 8]               0
           Conv2d-77            [2, 256, 8, 8]         589,824
      BatchNorm2d-78            [2, 256, 8, 8]             512
           Conv2d-79            [2, 256, 8, 8]         589,824
      BatchNorm2d-80            [2, 256, 8, 8]             512
             ReLU-81            [2, 256, 8, 8]               0
           Conv2d-82            [2, 256, 8, 8]         589,824
      BatchNorm2d-83            [2, 256, 8, 8]             512
           Conv2d-84            [2, 256, 8, 8]         589,824
      BatchNorm2d-85            [2, 256, 8, 8]             512
             ReLU-86            [2, 256, 8, 8]               0
           Conv2d-87            [2, 256, 8, 8]         589,824
      BatchNorm2d-88            [2, 256, 8, 8]             512
           Conv2d-89            [2, 256, 8, 8]         589,824
      BatchNorm2d-90            [2, 256, 8, 8]             512
             ReLU-91            [2, 256, 8, 8]               0
           Conv2d-92            [2, 256, 8, 8]         589,824
      BatchNorm2d-93            [2, 256, 8, 8]             512
           Conv2d-94            [2, 256, 8, 8]         589,824
      BatchNorm2d-95            [2, 256, 8, 8]             512
             ReLU-96            [2, 256, 8, 8]               0
           Conv2d-97            [2, 256, 8, 8]         589,824
      BatchNorm2d-98            [2, 256, 8, 8]             512
           Conv2d-99              [2, 2, 8, 8]             512
     BatchNorm2d-100              [2, 2, 8, 8]               4
            ReLU-101              [2, 2, 8, 8]               0
         Flatten-102                  [2, 128]               0
          Linear-103                   [2, 32]           4,128
            ReLU-104                   [2, 32]               0
          Linear-105                    [2, 1]              33
            Tanh-106                    [2, 1]               0
          Conv2d-107            [2, 256, 8, 8]         589,824
     BatchNorm2d-108            [2, 256, 8, 8]             512
            ReLU-109            [2, 256, 8, 8]               0
          Conv2d-110             [2, 73, 8, 8]         168,192
         Flatten-111                 [2, 4672]               0
         Softmax-112                 [2, 4672]               0
================================================================
Total params: 23,419,973
Trainable params: 23,419,973
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.05
Forward/backward pass size (MB): 25.47
Params size (MB): 89.34
Estimated Total Size (MB): 114.86
----------------------------------------------------------------
```

Acknowledgements

Our work was mostly based on Erwann Le Pennec’s teaching material; Sutton and Barto’s reference book “Reinforcement Learning”; and David Silver’s previous articles and teaching material available online.

Authorship

All authors contributed equally.
All authors critically reviewed and approved the final version of the manuscript.

References

1. Silver D, Hubert T, Schrittwieser J, et al. Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm. *ArXiv171201815 Cs*. Published online December 5, 2017. Accessed January 12, 2022. http://arxiv.org/abs/1712.01815

2. Sutton RS, Barto AG. *Reinforcement Learning, Second Edition: An Introduction*. Braford Books.; 2018.

3. Silver D, Hubert T, Schrittwieser J, et al. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science*. 2018;362(6419):1140-1144. doi:10.1126/science.aar6404