AML in depth study: Reinforcement Learning

Gianluca Giudice
University of Milano-Bicocca
g.giudice2@campus.unimib.it

January 31, 2022

Abstract

The aim of reinforcement learning is to build an agent, or decision maker, that learn to achieve a goal by interacting with the environment. This type of learning is performed through a sequence of rewards and punishments which depend on the decisions made by the agent. In reinforcement learning the agent aims to maximize the cumulative reward. In this work I'll give an introduction to reinforcement learning, starting from the fundamental aspects, until the latest deep reinforcement learning techniques. As an example of deep reinforcement learning I'll present and explain AlphaGo zero [1], one of the most famous application of reinforcement learning, where David Silver et al. managed to build an agent able to achieve superhuman performance in the game of Go starting without human knowledge. At the end of this work I'll develop an algorithm, based on AlphaGo zero, to master the game of Connect4 without human knowledge.

1 Introduction

Reinforcement learning is a computational approach to understanding and automating goal-directed learning and decision making. In this framework, learning is intended as mapping situations to actions so as to maximize a numerical reward signal. The learner is not told which actions to take, but instead must discover which actions yield the most reward, by trying them thorough interactions with the environment. In the most interesting and challenging cases, actions may affect not only the immediate reward but also the next situation and, through that, all subsequent rewards. Trial-error and delayed reward are the two most important distinguishing features of reinforcement learning [2].

Markov decision processes are used in the reinforcement learning framework to formally describe the environment. A learning agent must be able to sense the state of its environment and to take actions that affect the state in order to achieve a goal. Markov decision processes are useful because they include the fundamental features of a problem (sensation, action and goal) in order to capture cause and effect.

In many complex domains, reinforcement learning is the only feasible way to train a program in order to solve a problem. For example, in game playing, it is very hard for a human to provide accurate and consistent evaluations of board positions, which would be needed to define an evaluation function. Instead, the program can be told when it has won or lost, and it can use this information to learn an evaluation function that gives reasonably accurate estimates of the probability of winning from any given position. Similarly, it is extremely difficult to program an agent to fly a helicopter. On the other hand, given appropriate negative rewards, an agent can learn to fly by itself.[3]

2 Key Concepts in Reinforcement Learning

The main characters of RL are the agent and the environment. The environment is the world that the agent lives in and interacts with. At each time t, the agent receives the current state s_t and reward r_t. It then chooses an action a_t from the set of available actions, which is subsequently sent to the environment. The environment moves to a new state s_{t+1} and the reward r_{t+1} associated with the transition (s_t, a_t, s_{t+1}) is determined, as shown in Figure 1.

To better explain what RL is about, it is important to go through these key concepts: [4]

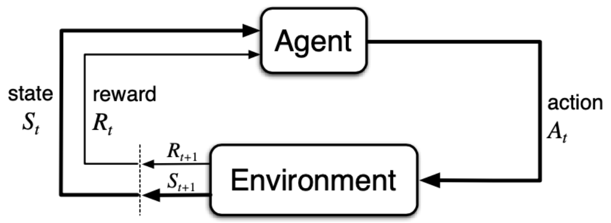


Figure 1: Typical RL scenario. [2]

- States and observations
- Action spaces
- Policies
- Reward and return
- RL optimization problem
- Value functions

2.1 States and observations

A **state** s is a complete description of the state of the world. There is no information about the world which is hidden from the state. An **observation** o is a partial description of a state, which may omit information on the world. When the agent is able to observe the complete state of the environment, we say that the environment is fully observed, otherwise, it is partially observed.

Markov decision processes (Markov Reward Processes plus actions), are a classical formalization of sequential decision making, where actions influence not only immediate rewards, but also future states, and through those, the future rewards. To this reason MDPs are a mathematical way to describe reinforcement learning problems.

Definition 1 *A* ***MDP*** *is a tuple* (S, A, P, R, γ)*, where:*

- *S is a (finite) set of Markov states $s \in S$*
- *A is a (finite) set of actions $a \in A$*
- *P is dynamics/transition model for each action, that specifies*
$$P(s_{t+1} = s' | s_t = s, a_t = a) \tag{1}$$
- *R is a reward function*
$$R(s_t = s, a_t = a) = \mathbb{E}[r_t | s_t = s, a_t = a] \tag{2}$$
- *γ is a discount factor $\gamma \in [0, 1]$*

2.2 Action spaces

Different environments allow different kinds of actions. The set of all valid actions in a given environment is called the **action space**. Some environments, like Atari and Go, have **discrete action spaces**, where only a finite number of moves are available to the agent. Other environments, like where the agent controls a robot in a physical world, have continuous action spaces. In continuous spaces, actions are real-valued vectors.

2.3 Policies

A **policy** defines the learning agent's way of behaving at a given time, it is a mapping from perceived states of the environment (i.e. observations) to actions to be taken when in those states. A policy fully defines the behavior of an agent:

Definition 2 *A* ***policy*** *π is a probability distribution over actions given states*

$$\pi(a|s) = \mathbb{P}[A_t = a | S_t = s] \tag{3}$$

The MDP and agent's policy together give rise to a **trajectory** τ, which is a sequence of states and actions in the world:

$$\tau = (s_0, a_0, r_0, s_1, a_1, r_1 ...)$$

2.4 Reward and return

The reward function R depends on the current state of the world and the action just taken:

$$r_t = R(s_t, a_t)$$

The agent aim to maximize the cumulative reward over a trajectory. The number of time-steps in the trajectory is the **horizon**, denoted by H, and can be infinite. The sum of all rewards obtained by the agent is discounted by how far off in the future they are obtained. The discount factor is $\gamma \in [0, 1]$. The cumulative reward over a trajectory is:

$$R(\tau) = \sum_{t=0}^{H-1} \gamma^t r_t \tag{4}$$

2.5 RL optimization problem

The goal in RL is to select a policy which **maximizes the expected return** when an agent acts according to it. The policy influences the probability distribution over trajectories. The probability of a T-step trajectory is:

$$P(\tau|\pi) = \rho_0(s_0) \prod_{t=0}^{T-1} P(s_{t+1}|s_t, a_t)\pi(a_t|s_t) \tag{5}$$

The expected return, denoted by $J(\pi)$, is then:

$$J(\pi) = \mathop{\mathbb{E}}_{\tau\sim\pi} [R(\tau)] \tag{6}$$

The central optimization problem in RL can then be expressed by:

$$\pi^* = \arg\max_{\pi} J(\pi) \tag{7}$$

with π^* being the optimal policy.

2.6 Value functions

It is very important to point out the expected return by starting in a specific state or state-action pair. These could be considered as functions and are a crucial aspect since they are used in almost every RL algorithm.

There are two main functions: [5]

1. The **State-Value Function** of a policy.
2. The **Action-Value Function** of a policy.

Definition 3 *The* ***State-Value Function*** *of a policy, $V^\pi(s)$ is the expected return by starting in state s and always acting according to policy π:*

$$V^\pi(s) = \mathop{\mathbb{E}}_{\tau\sim\pi} [R(\tau)|s_0 = s] \tag{8}$$

Definition 4 *The* ***Action-Value Function*** *of a policy, $Q^\pi(s, a)$ is the expected return by starting in state s and taking an arbitrary action a (which may not have come from the policy), and then forever after acting according to the policy π:*

$$Q^\pi(s, a) = \mathop{\mathbb{E}}_{\tau\sim\pi} [R(\tau)|s_0 = s, a_0 = a] \tag{9}$$

The state value function and action-value function are connected by the following relationship:

$$V^{\pi}(s) = \mathop{\mathbb{E}}_{a\sim\pi}[Q^{\pi}(s,a)] \tag{10}$$

Starting form these definitions it is possible to introduce two other key functions:

1. The **Optimal State-Value Function**.
2. The **Optimal Action-Value Function**.

2.7 Optimal value function and optimal policy

Value functions define a partial ordering over policies. A policy π is defined to be better than or equal to a policy π' if its expected return is greater than or equal to that of π' for all states. In other words, $\pi \geq \pi'$ if and only if $v_{\pi}(s) \geq v_{\pi'}(s)$ for all $s \in S$. There is always at least one policy that is better than or equal to all other policies. This is an optimal policy and it is denoted by π^*.

Definition 5 *The* ***Optimal State-Value Function****, $V^*(s)$ is the expected return by starting in state s and always acting according to the optimal policy in the environment:*

$$V^*(s) = \max_{\pi} \mathop{\mathbb{E}}_{\tau\sim\pi}[R(\tau)|s_0 = s] = \max_{\pi} V^{\pi}(s) \tag{11}$$

Definition 6 *The* ***Optimal Action-Value Function****, $Q^*(s,a)$ is the expected return by starting in state s, taking an arbitrary action a, and then forever after acting according to the optimal policy in the environment:*

$$Q^*(s,a) = \max_{\pi} \mathop{\mathbb{E}}_{\tau\sim\pi}[R(\tau)|s_0 = s, a_0 = a] = \max_{\pi} Q^{\pi}(s,a) \tag{12}$$

There is an important connection between the optimal action-value function $Q^*(s,a)$ and the action selected by the optimal policy.

By definition, $Q^*(s,a)$ gives the expected return for starting in state s, taking (arbitrary) action a, and then acting according to the optimal policy forever after. As a result, by having $Q^*(s,a)$, it is possible to directly obtain the optimal action $a^*(s)$:

$$a^*(s) = \arg\max_{a\in A} Q^*(s,a) \tag{13}$$

Therefore an optimal policy can be found by maximizing over $Q^*(s,a)$:

$$\pi^*(a|s) = \begin{cases} 1 & if\ a = \arg\max_{a\in A} Q^*(s,a) \\ 0 & otherwise \end{cases} \tag{14}$$

As seen for the state value function and action-value function, the following equation holds:

$$V^*(s) = \max_{a} Q^*(s,a) \tag{15}$$

2.8 Bellman equations for MRPs

The state-value function as well as the action-value function can be decomposed into the immediate reward plus the discounted value of the successor state, this is referred as Bellman equation. The Bellman equations for the state-value function and the action-value functions are:

$$V^{\pi}(s) = \mathop{\mathbb{E}}_{a\sim\pi}[\, r(s,a) + \gamma V^{\pi}(S_{t+1}) \mid S_t = s\,] \tag{16}$$

$$Q^{\pi}(s,a) = \mathbb{E}[\, r(s,a) + \gamma Q^{\pi}(S_{t+1}, A_{t+1}) \mid S_t = s, A_t = a\,] \tag{17}$$

The Bellman equations for the optimal value functions are:

$$V^*(s) = \max_a \mathbb{E}[\, r(s,a) + \gamma V^*(S_{t+1}) \mid S_t = s \,] \tag{18}$$

$$Q^*(s,a) = \mathbb{E}[\, r(s,a) + \gamma \max_a Q^*(S_{t+1}, A_{t+1}) \mid S_t = s, A_t = a \,] \tag{19}$$

The difference between the Bellman equations for the value functions relative to a policy and the optimal value functions, is the absence or presence of the max over actions. Its inclusion reflects the fact that whenever the agent gets to choose its action, in order to act optimally, it has to pick whichever action leads to the highest value.

3 RL algorithms

As seen before, from Equation 6 and Equation 7 the final objective of RL is to find the best policy, i.e.:

$$\pi^* = \arg\max_\pi \mathop{\mathbb{E}}_{\tau\sim\pi} [R(\tau)] \tag{20}$$

There are a variety of algorithms to do so (see Figure 2), depending on what to learn and how to learn it, for each of those, trade-offs must be taken into account.

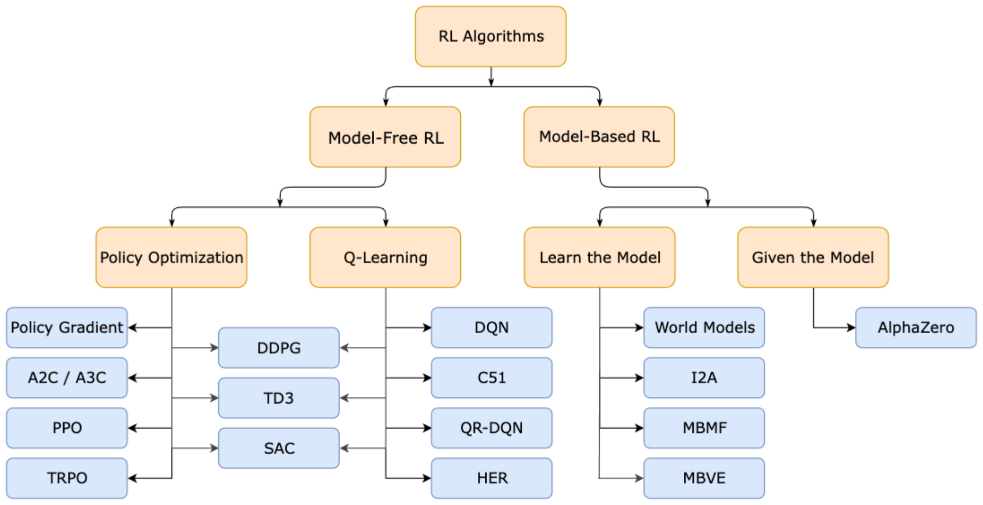


Figure 2: A non-exhaustive taxonomy of RL algorithms. [4]

3.1 Model-free vs model-based

When defining RL algorithms, the first branching point (as shown in Figure 2) is whether the agent has access to a model of the environment or needs to learn it.

Models are used for planning, meaning a 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 model-free methods that are explicitly trial-and-error learners.

The advantages of having a model are that this allows the agent to plan by thinking ahead, seeing what would happen for a range of possible choices, and explicitly deciding between its options. A famous example of this approach is AlphaZero. [1]

Even if having a model could really help the learner, unfortunately, a ground-truth model of the environment is not always available to the agent. In these cases agents have to learn the model directly from experience, and the model-learning is fundamentally hard.

3.2 What to learn

Another critical branching point in an RL algorithm is deciding what to learn, this includes:

- Policy
- Action-value function
- State-value function
- Model of the environment

Policy methods directly differentiate the objective function in Equation 20. In contrast, action-value function and state-value function methods, seeks to estimate the optimal state-function or Q-function of the optimal policy, but not explicitly the policy itself.

3.3 Q-learning

Q-Learning methods learn an approximation $Q_\theta(s,a)$ for the optimal action-value function, $Q^*(s,a)$. The Q-learning algorithm keeps an estimate $Q_t(s,a)$ of $Q^*(s,a)$ for each state-action pair $(s,a) \in S \times A$. Upon observing $(s_t, a_t, r_{t+1}, s_{t+1})$, the estimates are updated as follows [6]:

$$Q_{t+1}(s,a) = Q_t(s,a) + \alpha_t [\, r_{t+1} + \gamma \max_{a' \in A} Q_t(s_{t+1}, a') - Q_t(s_t, a_t)\,] \tag{21}$$

Q has been shown to converge with probability 1 to Q^* [2]. The Q-learning algorithm is shown below in procedural form.

Algorithm 1 Q-learning for estimating $\pi \approx \pi^*$

```
Require:
  Learning rate $\alpha \in (0,1]$
  procedure QLEARNING($\mathcal{X}$, $A$, $R$, $T$, $\alpha$, $\gamma$)
    Initialize $Q(s,a)$, for all $s \in S$, $a \in A$, arbitrarily except that $Q(terminal, \cdot) = 0$
    while $s$ is not terminal do
      Choose $a$ from $s$ using policy derived from $Q$
      Take action $a$, observe $r$, $s'$
      $Q(s,a) \leftarrow Q(s,a) + \alpha[\cdot(r + \gamma \max_a Q(s',a) - Q(s,a))]$
      $s \leftarrow s'$
    end while
  end procedure
```

In this case, the learned action-value function Q, directly approximates the optimal action-value function Q^*. Then from the learned action-value function that approximate Q^* it is possible to derive π^* by Equation 14.

3.4 Function approximation

Q-learning algorithm is a method to achieve optimal value functions and optimal policies. An agent that learns an optimal policy reaches its goal, but in practice this rarely happens. For the most interesting tasks, optimal policies can be generated only with extreme computational cost. Even if the agent has a complete and accurate model of the environment's dynamics, it is usually not possible to simply compute an optimal policy by solving the Bellman optimality equation. A critical aspect of the problem faced by the agent is always the computational power available to it, in particular, the amount of computation it can perform in a single time step.

The memory available is also an important constraint. A large amount of memory is often required to build up approximations of value functions, policies, and models. In the Q-learning algorithm (Algorithm 1.), it is necessary to build a table for each state-action pair, however, in many cases of practical interest there are far more states that could possibly be entries in a table, so this approach is not scalable. In these cases the functions must be approximated, using some sort of more compact parameterized function representation.

Besides the space and time complexity when dealing with a huge number of states, it could be useful to achieve some approximations for other reasons. For example, in approximating optimal behavior, there may be many states that the agent faces with such a low probability. The online nature of reinforcement learning (i.e. data becomes available in a sequential order over time) makes it possible to approximate optimal policies in ways that put more effort into learning to make good decisions for frequently encountered states, at the expense of less effort for infrequently encountered states. This is one key property that distinguishes reinforcement learning from other approaches to approximately solving MDPs.

To make sensible decisions in large states space it is necessary to generalize from previous encountered states. Those previous encountered states are different from the current ones but could be in some sense similar. The kind of generalization required is called function approximation because it takes examples from a desired function (e.g. a value function) and attempts to generalize from them to construct an approximation of the entire function. To achieve this, any of a broad range of existing methods for supervised-learning function approximation, can be used simply by treating each update as a training example provided by interaction between the agent and the environment.

3.5 Deep RL

Since achieving generalization in state representation is simply function approximation, artificial neural networks could be used to approximate these functions. Most successful RL applications that deal with a huge amount of states, rely on hand-crafted features combined with linear value functions or policy representations to approximate states [7]. The performance of such systems heavily relies on the quality of the feature representation.

In the work of David Silver et al. "Playing Atari with Deep Reinforcement Learning" [7] they presented a deep learning model to successfully learn control policies directly from high-dimensional sensory input using reinforcement learning. The idea is to represent $Q(s, a)$ as a neural network $f(s)$, which given a vector s will output a vector of Q-values for all possible actions a, as shown in Figure 3.

In the work "Playing Atari with Deep Reinforcement Learning" [7], they developed an agent on the challenging domain of classic Atari 2600 games. They demonstrate that the deep Q-network agent, receiving only the pixels (processed by a deep convolutional neural network) and the game score as inputs, was able to surpass the performance of all previous algorithms. Furthermore it achieve a level comparable to that of a professional human games tester. This work bridges the divide between high-dimensional sensory inputs and actions, resulting in the first artificial agent that is capable of learning to excel at a diverse array of challenging tasks.

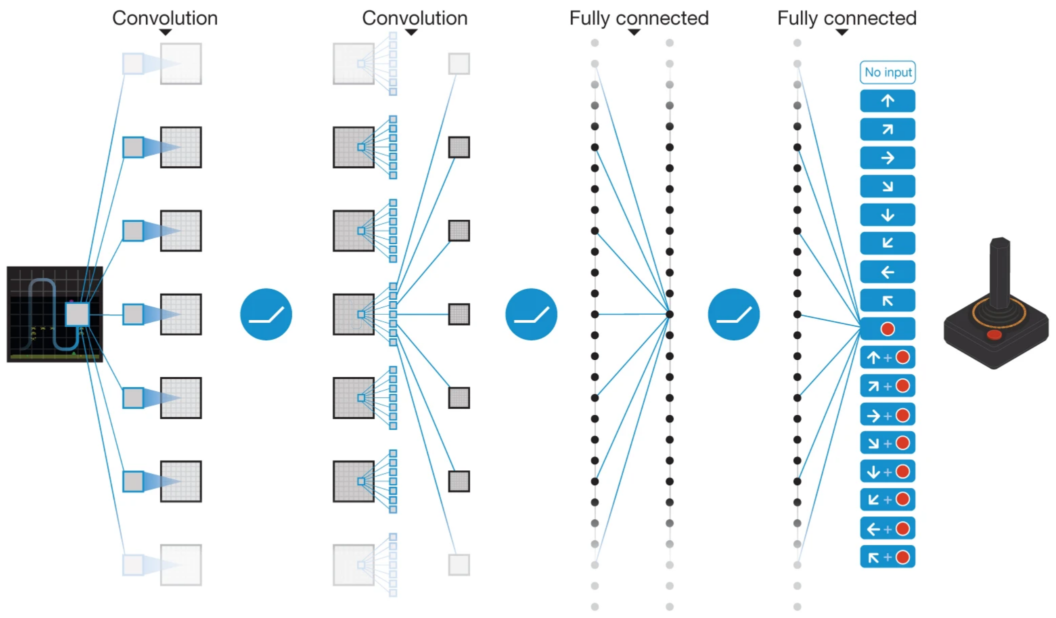


Figure 3: Architecture of the neural network used to predict actions. [7].

4 Case study: AlphaGo zero

4.1 Motivation

The game of Go has long been viewed as the most challenging of classic games for artificial intelligence owing to its enormous search space and the difficulty of evaluating board positions and moves. In the Table 1 the complexity of some famous board games is summarized:

Table 1: Complexity of the most famous board game.

Game	Board size (positions)	State-space complexity (as log to base 10)	Game-tree complexity (as log to base 10)	Ref.	Complexity class
Tic-tac-toe	9	3	5		PSPACE-complete [8]
Connect Four	42	13	21	[9]	in PSPACE
Checkers	32	20 or 18	40	[9]	EXPTIME-complete [10]
Chess	64	44	123	[11]	EXPTIME-complete [12]
Shogi	81	71	226	[13]	EXPTIME-complete [13]
Go (19x19)	361	170	360	[9]	EXPTIME-complete [14]

As described in the work "Mastering the game of Go with deep neural networks and tree search"[15] by David Silver et al., all games of perfect information have an optimal value function $v^*(s)$, which determines the outcome of the game, from every board position or state s, under perfect play by all players.

These games may be solved by recursively computing the optimal value function in a search tree containing approximately b^d **possible sequences of moves**, where b **is the game's breadth** (number of legal moves per position) and d **is its depth** (game length). In large games, such as chess ($b \approx 35$,$d \approx 80$) [9] and especially Go ($b \approx 250$, $d \approx 150$) [9], exhaustive search is infeasible, but the effective search space can be reduced by two general principles:

1. **The depth of the search may be reduced** by position evaluation: truncating the search tree at state s and replacing the subtree below s by an approximate value function $v(s) \approx v^*(s)$ that predicts the outcome from state s. This approach has led to superhuman performance in chess [16] (the famous "Deep Blue" program that defeated Garry Kasparov[16]) and checkers [17] but it was believed to be intractable in Go due to the complexity of the game [18].
2. **The breadth of the search may be reduced** by sampling actions from a policy $p(a|s)$ that is a probability distribution over possible moves a in position s. For example, Monte Carlo rollouts search to maximum depth without branching at all, by sampling long sequences of actions for both players from a policy p. Averaging over such rollouts can provide an effective position evaluation, achieving weak amateur level play in Go [19].

4.2 Versions of AlphaGo

The first version of AlphaGo [15] achieved a 99.8% winning rate against other Go programs, and defeated the human European Go champion by 5 games to 0. That was the first time that a computer program has defeated a human professional player in the full-sized game of Go. This was achieved by a combination of:

- Supervised learning from human expert games
- Reinforcement learning from games of self-play
- Monte Carlo simulation with value and policy networks

The next version of AlphaGo is AlphaGo Zero [1], an algorithm based solely on reinforcement learning, without human data, guidance or domain knowledge beyond game rules. Starting tabula rasa, AlphaGo Zero achieved superhuman performance in Go, winning 100-0 against the previously published, champion-defeating, AlphaGo.

4.3 Neural network architecture of AlphaGo Zero

In AlphaGo Zero a deep neural network f_θ with parameters θ is used. This neural network takes as an input the raw board representation s of the position and outputs both move probabilities and a value, $f_\theta = (p, v)$.

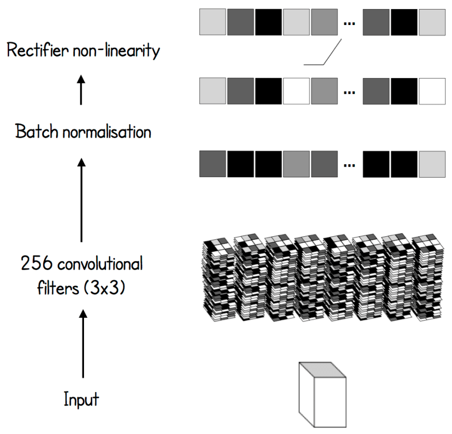
The output of the neural networks are:

- Vector p: the vector of move probabilities p represents the probability of selecting each move $p_a = P(a|s)$
- Scalar v: the value v is a scalar evaluation, estimating the probability of the current player winning from position s

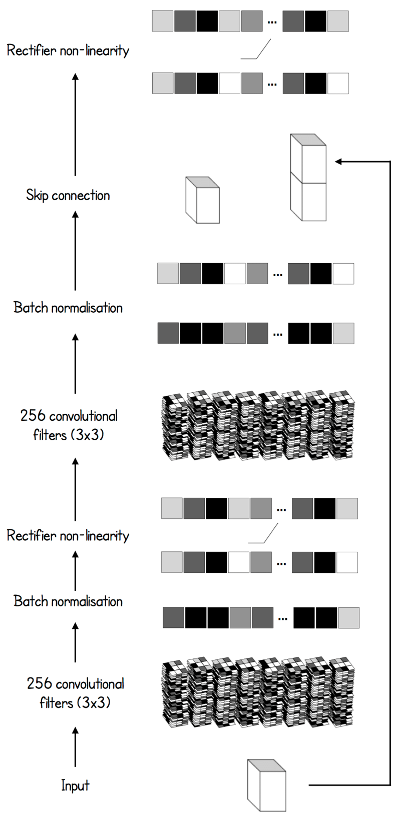
In the previous version of AlphaGo [15], two different deep neural networks were used to predict both p and v. On the other hand, in AlphaGo Zero, these two neural networks are combined into a single one with two different outputs, treating the task as a multi-task learning problem. Another new aspect of this neural network is the introduction of many residual blocks of convolutional layers.

The final neural network takes as input features the board state s, and consists in these layers:

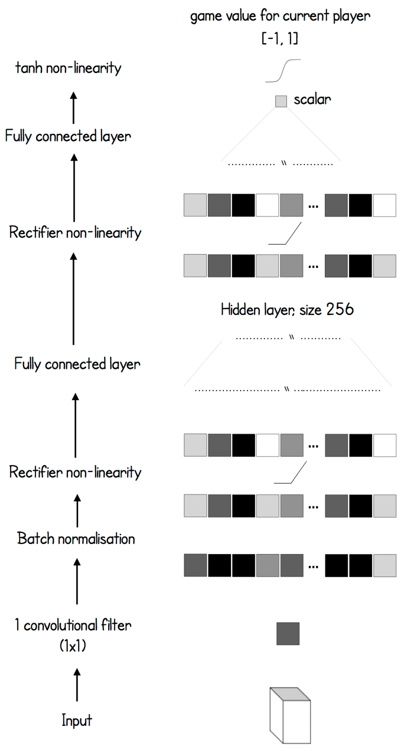
1. First convolutional block:
   (a) A convolution of 256 filters of kernel size 3×3 with stride 1
   (b) Batch normalization
   (c) A rectifier nonlinearity
2. 39 residual blocks, each one applies the following modules sequentially to its input:
   (a) A convolution of 256 filters of kernel size 3×3 with stride 1
   (b) Batch normalization
   (c) A rectifier nonlinearity
   (d) A convolution of 256 filters of kernel size 3×3 with stride 1
   (e) Batch normalization
   (f) A skip connection that adds the input to the block
   (g) A rectifier nonlinearity
3. The output of the residual tower is passed into two separate "heads" for computing the policy and value:
   - The policy head:
     (a) A convolution of 2 filters of kernel size 1×1 with stride 1
     (b) Batch normalization
     (c) A rectifier nonlinearity
     (d) A fully connected linear layer that outputs a vector of size $19^2 + 1 = 362$ ($19 * 19$ possible actions +1 for the pass acton)
   - The value head:
     (a) A convolution of 1 filter of kernel size 1×1 with stride 1
     (b) Batch normalization
     (c) A rectifier nonlinearity
     (d) A fully connected linear layer to a hidden layer of size 256
     (e) A rectifier nonlinearity
     (f) A fully connected linear layer to a scalar
     (g) A tanh nonlinearity outputting a scalar in the range $[-1, 1]$ (1 representing the current player winning the game, -1 for the specular case)



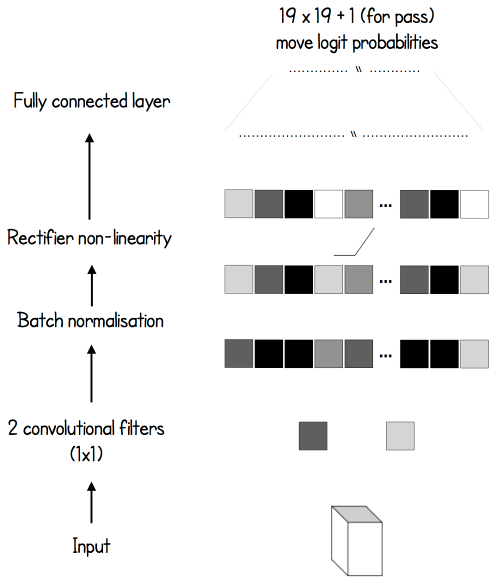
(a) The **convolutional layer.**



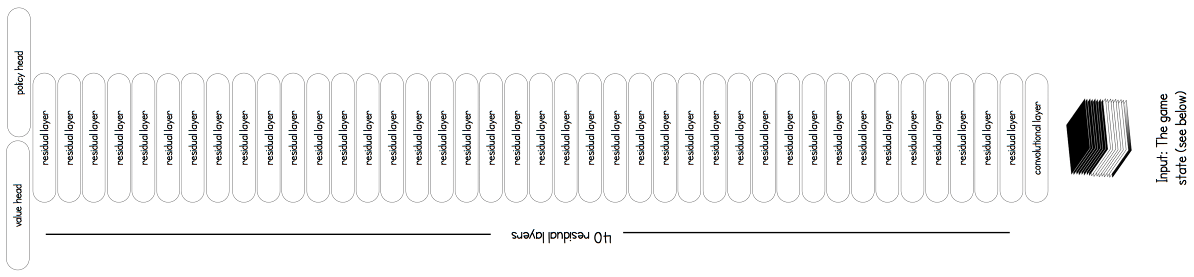
(b) A **residual layer.**



(c) The **value head.**



(d) The **policy head.**



(e) **Full neural network.**

Figure 4: The architecture of AlphaGo Zero. Diagram from this article.[1]

4.4 Reinforcement learning in AlphaGo Zero

The neural network in AlphaGo Zero is trained from games of self-play by a novel reinforcement learning algorithm. In each position s, a MCTS search is executed, guided by the neural network f_θ. The MCTS search outputs probabilities π of playing each move. These search probabilities usually select much stronger moves than the raw move probabilities p of the neural network $f_\theta(s)$; MCTS may therefore be viewed as a powerful policy improvement operator. Self-play with search may be viewed as a powerful policy evaluation operator. During the self-play the MCTS-based policy is used to select each move, and the game winner z is used as a sample of the value v (the winning player).

4.4.1 Self-play

The main idea of this reinforcement learning algorithm is to use these search operators in a policy iteration procedure: the neural network's parameters are updated to make the move probabilities and value $(p, v) = f_\theta(s)$ more closely match the improved search probabilities and self play winner (π, z). These new parameters are used in the next iteration of self-play to make the search even stronger.

The self-play pipeline is performed as following (see Figure 5): the program plays a game s_1, ..., s_t against itself. In each position s_t, a MCTS α_θ is executed (see Figure 7) using the latest neural network f_θ. Moves are selected according to the search probabilities computed by the MCTS, $a_t \sim \pi_t$. The terminal position s_T is scored according to the rules of the game to compute the game winner z.

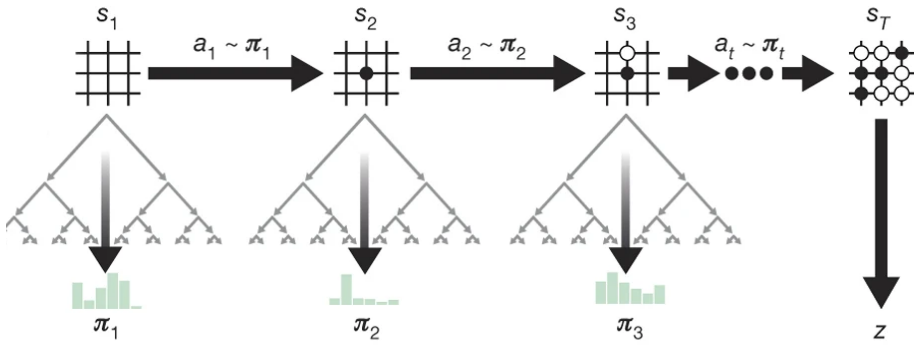


Figure 5: **Self-play** AlphaGo Zero.[15]

During the self-play procedure, each tuple (s_t, π_t, z) is stored as a training example for the training phase. During the training, the neural network takes the raw board position s_t as input, passes it through many convolutional layers with parameters θ, and outputs both a vector p_t, representing a probability distribution over moves, and a scalar value v_t, representing the probability of the current player winning in position s_t (see Figure 7). The neural network parameters θ are updated to maximize the similarity of the policy vector p_t to the search probabilities π_t, and to minimize the error between the predicted winner v_t and the game winner z (see Equation 24). The new parameters are used in the next iteration of self-play as shown in Figure 5.

[1]https://medium.com/applied-data-science/alphago-zero-explained-in-one-diagram-365f5abf67e0

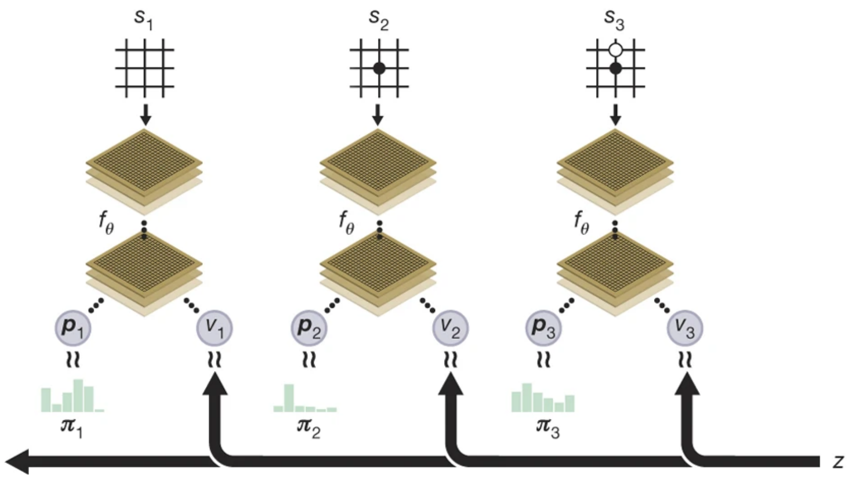


Figure 6: **Neural network training** in AlphaGo Zero.[15]

4.4.2 MCTS

The MCTS uses the neural network f_θ to guide its simulations (see Figure 7). Each edge (s, a) in the search tree stores a prior probability $P(s, a) = p_\theta(a|s)$, a visit count $N(s, a)$, and an action-value $Q(s, a)$.

Each simulation starts from the root state and iteratively selects moves $a_t = \arg\max_a Q(s_t, a) + U(s_t, a)$, until a leaf node s' is encountered. This leaf position is expanded and evaluated only once by the network to generate both prior probabilities and evaluation, $(P(s', \cdot), V(s')) = f_\theta(s')$.

The action-value function is defined as:

$$Q(s, a) = \frac{1}{N(s, a)} \sum_{s' \mid s, a \to s'} V(s') \tag{22}$$

where $s, a \to s'$ indicates that a simulation eventually reached s' after taking move a from position s.

The reason why the chosen action a maximize the sum $Q(s_t, a) + U(s_t, a)$ and not only the action-value function $Q(s_t, a)$ is because $U(s_t, a)$ acts as an exploration factor, a crucial aspect in reinforcement learning. The exploration-exploitation tradeoff is taken into account by this factor $U(s_t, a)$, which is defined as:

$$U(s, a) = c_{puct} P(s, a) \frac{\sum_b N(s, b)}{1 + N(s, a)} \tag{23}$$

where c_{puct} is a constant determining the level of exploration. This search control strategy initially prefers actions with high prior probability and low visit count, but asymptotically prefers actions with high action value.

MCTS may be viewed as a self-play algorithm that, given neural network parameters θ and a root position s, computes a vector of search probabilities recommending moves to play, $\pi = \alpha_\theta(s)$.

The key points of the algorithm are summarized in the Figure 7.

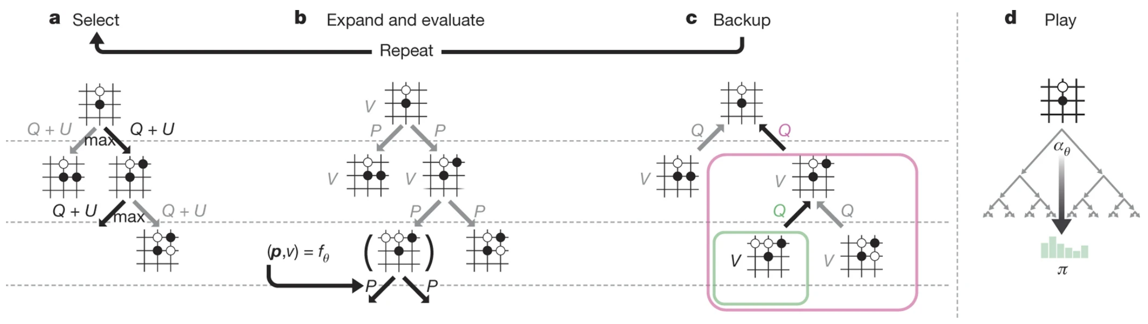


Figure 7: **MCTS procedure** in AlphaGo Zero[15]:
(a) Each simulation traverses the tree by selecting the edge with maximum action value Q, plus an upper confidence bound U that depends on a stored prior probability P and visit count N for that edge (which is incremented once traversed).
(b) The leaf node is expanded and the associated position s is evaluated by the neural network $(P(s,\cdot), V(s)) = f_\theta(s)$.
(c) Action-value Q is updated to track the mean of all evaluations V in the subtree below that action.
(d) Once the search is complete, search probabilities π are returned.

4.4.3 Loss function

The parameters θ are adjusted by gradient descent on a loss function L that sums over the mean-squared error to have v as close as possible to z, and cross-entropy losses to have p as close as possible to π:

$$f_\theta(s) = (p, v), \quad L = (z - v)^2 - \pi_t log(p) + c\,||\theta||^2 \tag{24}$$

where c is a parameter controlling the level of L2 weight regularization to prevent overfitting.

4.4.4 Training method

The training consists in N iterations, in which K self-play games are played by the player α_θ using MCTS guided by the neural network f_θ. In these games the player α_θ plays against itself in order to collect new data. In each K game, for every move to choose, J simulations of MCTS are performed in order to collect more confident outcomes of the games.

Once the new training examples are collected, the neural network is trained for a fixed number of epochs using the new data collected in the K self-play games of the N-th iteration, and the data from the previous iterations.

Using this method, the loss of the neural network is going to decrease and eventually leading to better search in the MCTS performed by α_θ, hence leading to higher quality training examples in the next iteration, and so on.

4.4.5 Evaluation

Since the MCTS is guided by the neural network f_θ, we want to be sure that the neural network f_θ is improving over the iterations. To this purpose, starting with the best neural network so far $f_{\theta'}$, the latest neural network is evaluated at the end of each iteration after the training with the new data.

In order to evaluate the neural network, two players α_θ and $\alpha_{\theta'}$ play P games against each other. If the percentage of the games won by the player $\alpha_{\theta'}$ is higher than a threshold, then the new neural network $f_{\theta'}$ is saved and considered the best network so far.

In many games, such as Go, the one who start the game leads to a small advantage in the match, to overcome this problem, the number of games for evaluation are performed letting each player starts the game half of the times.

4.5 Superhuman performance of AlphaGo Zero

As discussed in previous section, AlphaGo Zero has been trained without human knowledge, this eventually leads to superhuman performance in the game of Go, even beating 100-0 the previous version of AlphaGo. This previous version was trained using supervised learning on the master Go player moves. On the other hand, AlphaGo Zero started from completely random behavior and continued without human data nor intervention.

In order to understand the performance of AlphaGo Zero, it is interesting to analyze its empirical training results of each MCTS player α_{θ_i} from each iteration i of reinforcement learning in AlphaGo Zero, in comparison to a similar player trained by supervised learning from human data, using the KGS dataset[2]. The Elo rating is a way to define the strength of the player. The results are shown in Figure 8. As we can see, RL-approach is able to reach higher Elo rating and even in less amount of time.

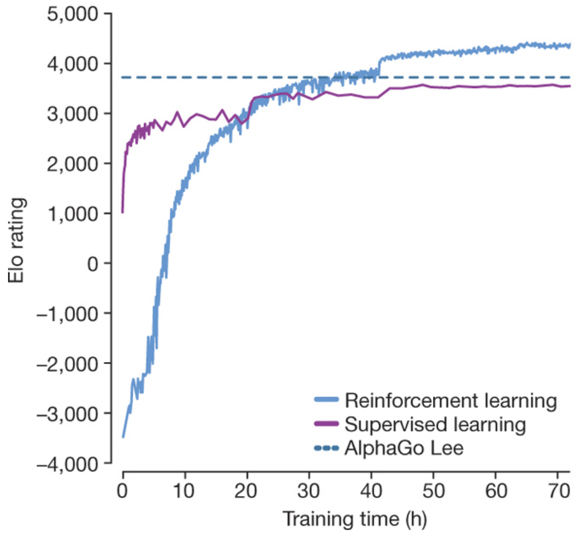


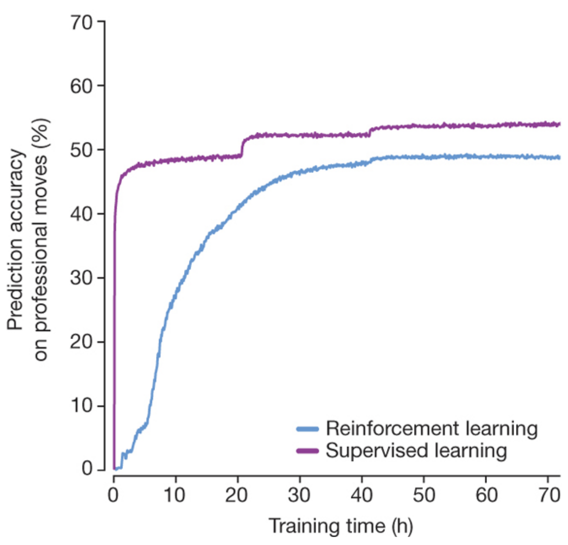
Figure 8: Elo rating comparison. [15]

Another interesting result is the difference in prediction accuracy on human professional moves between the neural network f_{θ_i} at each iteration of self-play i, and the neural network trained by supervised learning. The accuracy measures the percentage of positions in which the neural network assigns the highest probability of action to that taken by a professional human player, shown in Figure 9a.

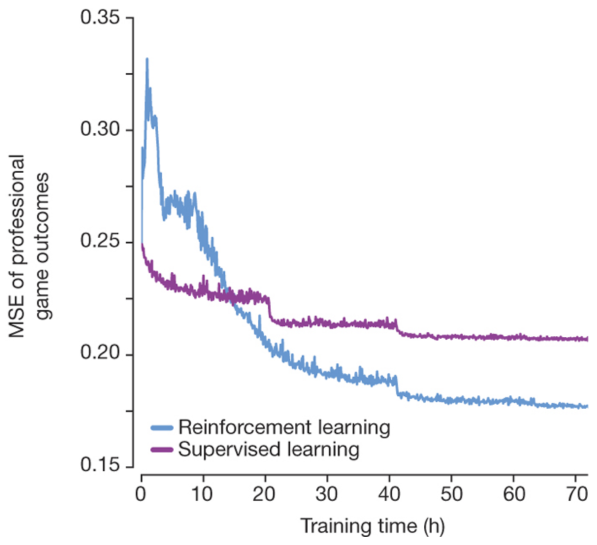
In addition to the accuracy in prediction human moves, it is possible to consider the MSE in predicting the outcome of games. The MSE is between the actual outcome $z \in -1, 1$ and the neural network value v, scaled to the range of 0-1, shown in Figure 9b.

The astonishing fact is that, even if AlphaGo zero performs worse in predicting master Go player moves, it is better in predict the actual outcome of a game (lower MSE). This is due to nature of reinforcement learning. Since in the RL approach we are not in charge of giving any kind of human expertise to the algorithm, there is no human knowledge ceiling to the performances.

[2]KGS dataset (available from `https://u-go.net/gamerecords/`).



(a) Prediction accuracy on human professional moves.



(b) MSE of human professional game outcomes.

Figure 9: AlphaGo Zero performance comparison. [15]

5 Hands-on: Connect4 zero

In this last section I will introduce Connect4 zero. Starting from the theory of RL and using the techniques of AlphaGo Zero [1], I will develop an algorithm based on RL that without human knowledge achieves a master level in the game of Connect4.

The reason why I have chosen Connect4 rather then other games, is due to the Table 1. As we can see from the table, the game Connect4 is relatively easy and could possibly be solved by tree-search and alpha-beta pruning. However, the point of my work is to give a proof of concept of the RL algorithm used in AlphaGo Zero without having the computational resources of DeepMind, which were necessary to train the agent for the Game of Go.

The algorithm that I developed is based on this code[3]. The code provided in the GitHub repository is only the starting point, since I changed it in order to support the game of Connect4, also introducing the possibility to play against the RL agent in a graphical user interface.

Besides this marginal change, I performed two major changes to the original code:

1. New neural network architecture.
2. Episodes of self-play performed in parallel.

The code that I developed is available here[4] on GitHub.

5.1 Proposed neural network architecture

Since the game of Connect4 is easier compared to the game of Go Table 1, it would be a waste of resources to use the same neural network proposed in subsection 4.3. For this reason I used a simpler architecture and I removed the skip-connections among layers. The neural network used consists in 4.5m parameters and is composed by:

1. Input Board (6×7 layer)
2. 3 convolutional blocks, each one composed by:
   (a) A convolution of 128 filters of kernel size 3×3 with stride 1
   (b) Batch normalization
   (c) A rectifier nonlinearity
3. The last convolution is flattened

[3]`https://github.com/suragnair/alpha-zero-general`
[4]`https://github.com/gianlucagiudice/alpha-zero-general`

4. First dense block composed by:
    (a) A fully connected layer of 256 nodes
    (b) Batch normalization
    (c) A rectifier nonlinearity
    (d) Dropout with factor 0.4
5. Second dense block composed by:
    (a) A fully connected layer of 128 nodes
    (b) Batch normalization
    (c) A rectifier nonlinearity
    (d) Dropout with factor 0.4
6. Two different "heads" for both policy and value
    - The policy head:
      (a) A fully connected layer
      (b) Softmax activation
    - The value head:
      (a) A fully connected layer
      (b) A tanh nonlinearity

5.2 Parallelization of self-play episodes

The second major change to the original code consists in the parallelization of the self-play episodes. For each iteration, the agent performs *N* games against itself using the MCTS, since these self-play games are independent to each-other, it is possible to perform them in parallel. The introduction of this computational parallelism leads to a dramatic decrease in time for the execution of each episode. I have taken inspiration from the work of David Silver "Playing Atari with Deep Reinforcement Learning" [7], where in order to decrease the training time, several agents run in parallel to collect training examples.

5.3 Hyperparameters

According to subsubsection 4.4.4, we need to choose the training hyperparameters. These parameters consist in:

- **Number of iterations**: 150
- **Self-play games per each iteration**: 100
- **Number of MCTS**: 25 (this parameter specifies the number of MCTS simulations performed by the agent in order to select the next move and generate new training examples)
- **Number of games against previous agent**: 10 (the number of games performed against an agent using the previous version of the neural network for evaluation purposes)
- **Update threshold**: 60% (if the agent using the new network is able to won al least 60% of the games, the new neural network is considered to be the best so far)
- **Number of training epochs**: 10 (the number of training epochs to fit the new and old data)

5.4 Training analysis

In the Figure 10 we can analyze the training history of the network neural network. As we can see, the loss slowly decrease through epochs, this means that the network is actually getting better in both predicting outcomes of the games and acting optimally.

The "stair pattern" of the loss, is due to the evaluation process. Indeed, when this happens we have just accepted the new model as the best model so far (as shown in Figure 11). Since the MCTS in the next iteration is guided by the new neural network, which is better than the previous one, this leads to a better search in the MCTS, therefore collecting better quality training examples.

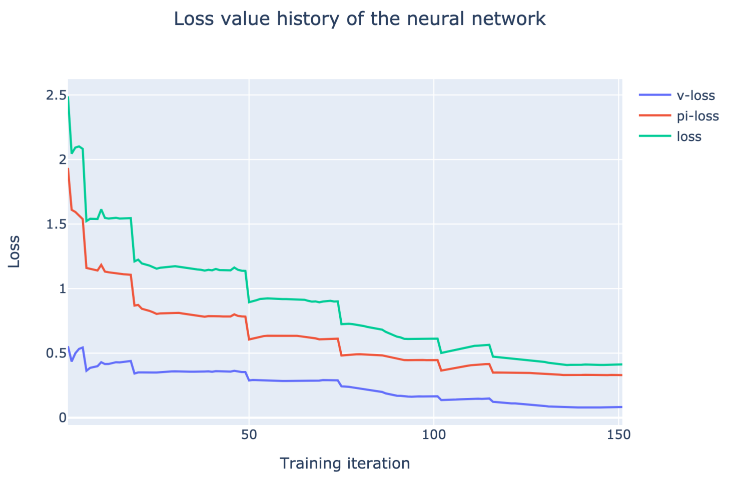


Figure 10: Training history of the model.

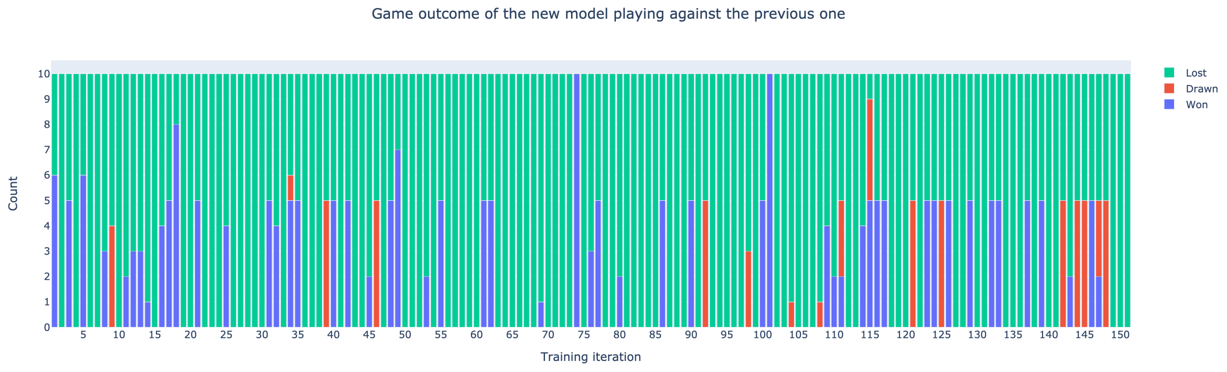


Figure 11: Pitting history of the model.

6 Conclusions

In this work, reinforcement learning is applied in the domain of board games, showing how, in a very challenging domain (such as the game of Go), this approach achieves superhuman performances. The most astonishing fact is that this level has been achieved without human knowledge (except for what concerns the rules of the game). The agent learned the best strategies by itself. This RL approach does not use exhaustive search techniques, it rather let the agent learn by trial and error.

Reinforcement learning is a very promising field. In this work, board games are used as a proxy to better understand and explore this approach. However, besides board games, the concept behind reinforcement learning is deeper than simply playing Go or Connect4. As discussed in the work "Reward is enough"[20] by David Silver et al. they hypothesise that the maximization of total reward may be enough to understand intelligence and its associated abilities. They conjecture that intelligence could emerge in practice from sufficiently powerful reinforcement learning agents that learn to maximize future reward.

References

[1] David Silver et al. “Mastering the game of go without human knowledge”. In: *nature* 550.7676 (2017), pp. 354–359.

[2] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: A Bradford Book, 2018. ISBN: 0262039249.

[3] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: a modern approach*. 3rd ed. Pearson, 2009.

[4] Joshua Achiam. “Spinning Up in Deep Reinforcement Learning”. In: (2018).

[5] David Silver. *Lectures on Reinforcement Learning*. URL: `https://www.davidsilver.uk/teaching/`. 2015.

[6] Csaba Szepesvári. *Algorithms for Reinforcement Learning*. Synthesis Lectures on Artificial Intelligence and Machine Learning. 2010. URL: `http://dx.doi.org/10.2200/S00268ED1V01Y201005AIM009`.

[7] Volodymyr Mnih et al. “Playing Atari with Deep Reinforcement Learning”. In: (2013). cite arxiv:1312.5602Comment: NIPS Deep Learning Workshop 2013. URL: `http://arxiv.org/abs/1312.5602`.

[8] Stefan Reisch. “Hex is PSPACE-complete”. In: *Acta Informatica* (1981).

[9] L. Victor Allis. “Searching for solutions in games and artificial intelligence”. In: 1994.

[10] John Michael Robson. “N by N Checkers is Exptime Complete”. In: *SIAM J. Comput.* 13 (1984), pp. 252–267.

[11] Claude E. Shannon. “XXII. Programming a computer for playing chess”. In: *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* 41.314 (1950), pp. 256–275. DOI: `10.1080/14786445008521796`. eprint: `https://doi.org/10.1080/14786445008521796`. URL: `https://doi.org/10.1080/14786445008521796`.

[12] Aviezri S Fraenkel and David Lichtenstein. “Computing a perfect strategy for n × n chess requires time exponential in n”. In: *Journal of Combinatorial Theory, Series A* 31.2 (1981), pp. 199–214. ISSN: 0097-3165. DOI: `https://doi.org/10.1016/0097-3165(81)90016-9`. URL: `https://www.sciencedirect.com/science/article/pii/0097316581900169`.

[13] Hiroyuki Iida, Makoto Sakuta, and Jeff Rollason. “Computer shogi”. In: *Artificial Intelligence* 134.1 (2002), pp. 121–144. ISSN: 0004-3702. DOI: `https://doi.org/10.1016/S0004-3702(01)00157-6`. URL: `https://www.sciencedirect.com/science/article/pii/S0004370201001576`.

[14] John Robson. “The Complexity of Go.” In: vol. 9. Jan. 1983, pp. 413–417.

[15] David Silver et al. “Mastering the Game of Go with Deep Neural Networks and Tree Search”. In: *Nature* 529.7587 (Jan. 2016), pp. 484–489. DOI: `10.1038/nature16961`.

[16] Murray Campbell, A.Joseph Hoane, and Feng-hsiung Hsu. “Deep Blue”. In: *Artificial Intelligence* 134.1 (2002), pp. 57–83. ISSN: 0004-3702. DOI: `https://doi.org/10.1016/S0004-3702(01)00129-1`. URL: `https://www.sciencedirect.com/science/article/pii/S0004370201001291`.

[17] Michael Buro. “From Simple Features to Sophisticated Evaluation Functions”. In: *Computers and Games*. Ed. by H. Jaap van den Herik and Hiroyuki Iida. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 126–145. ISBN: 978-3-540-48957-3.

[18] Martin Müller. “Computer Go”. In: *Artificial Intelligence* 134.1 (2002), pp. 145–179. ISSN: 0004-3702. DOI: `https://doi.org/10.1016/S0004-3702(01)00121-7`. URL: `https://www.sciencedirect.com/science/article/pii/S0004370201001217`.

[19] B. Bouzy and B. Helmstetter. “Monte-Carlo Go Developments”. In: *Advances in Computer Games: Many Games, Many Challenges*. Ed. by H. Jaap Van Den Herik, Hiroyuki Iida, and Ernst A. Heinz. Boston, MA: Springer US, 2004, pp. 159–174. ISBN: 978-0-387-35706-5. DOI: `10.1007/978-0-387-35706-5_11`. URL: `https://doi.org/10.1007/978-0-387-35706-5_11`.

[20] David Silver et al. “Reward is enough”. In: *Artificial Intelligence* 299 (2021), p. 103535. ISSN: 0004-3702. DOI: `https://doi.org/10.1016/j.artint.2021.103535`. URL: `https://www.sciencedirect.com/science/article/pii/S0004370221000862`.