The use of reinforcement learning to discover new mathematical algorithms

David Rosado
Universitat de Barcelona

Àlex Pujol
Universitat de Barcelona

Abstract

Reinforcement learning is an area of machine learning concerned with how intelligent agents ought to take actions in an environment in order to maximize the notion of cumulative reward. In this paper, we will study a deep reinforcement learning approach based on AlphaZero for discovering efficient and provably correct algorithms for the multiplication of arbitrary matrices.

1 Introduction

Machine learning algorithms are a reality in today's context. The use of supervised and unsupervised learning to approach problems in which there is available data and a pattern, is extremely useful and in many cases successful. However, usually the data needed to solve a problem exceeds tremendous amounts, which makes the task roughly expensive and practically inefficient. Reinforcement learning (RL) overcomes this issue since it consists in a decision-making system trained to make optimal decisions. The algorithm that does not rely directly on data, but learns from the interaction with the environment, either real or simulated.

Such models are based on training a set of agents to make a series of decisions in the predefined environment. The agents learn through trial and error, by taking actions and receiving feedback in the form of rewards, or reinforcements, and penalties. The goal for those agents is to learn a policy that will allow them to maximize the cumulative reward over timer, or minimize the punishment. Let us show an outline of RL, see Figure 1.

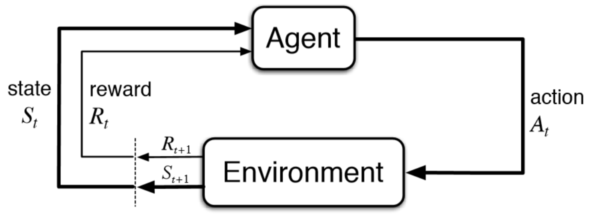


Figure 1: Reinforcement learning outline

We can find three main categories within RL [7](Àlex): value-based methods, policy-based methods and model-based methods. The first one consists on estimating a value function that predicts the expected reward by taking a particular action on a given state. Different action pathways must be explored under simulation and the agent chooses the action that leads to maximize the value function. Examples of value-based reinforcement learning algorithms include Q-learning. Policy-based methods consist on directly learning the stochastic policy function that maps the given state of the environment to the action of the agent. In other words, the agent learns a policy that tells it what action needs to do at every given state. Examples of policy-based reinforcement learning algorithms include REINFORCE. Many agents with different policies need to be sampled. Finally, model-based algorithms involve learning a model of the environment by observing the state transitions and rewards that result from different actions, and using it to plan a sequence of actions that will maximize the expected reward. Examples of model-based reinforcement learning algorithms include Monte Carlo Tree Search.

Every of the previous approaches to RL has its strengths and flaws [6](Àlex and David). On one hand, value-based methods are typically easier to implement and tune than policy-based methods, and they can be more sample efficient, meaning they may require fewer data to learn effectively. They are also generally easier to interpret and understand, as the value function provides a clear indication of the expected return for each action. However, value-based methods can be inefficient when the state space is large or when the optimal policy is stochastic (non-deterministic). Policy-based methods, on the other hand, can handle these types of environments more naturally, converging easier and quicker. Model-based reinforcement learning algorithms can be very sample efficient, as they can use a learned model of the environment to plan actions. They are also well-suited to environments where the transition dynamics are complex or uncertain. However, learning a good model of the environment can be challenging, and model-based algorithms may require a lot of data to learn an accurate model. In addition, these algorithms can be computationally expensive, as they often involve planning over numerous time steps.

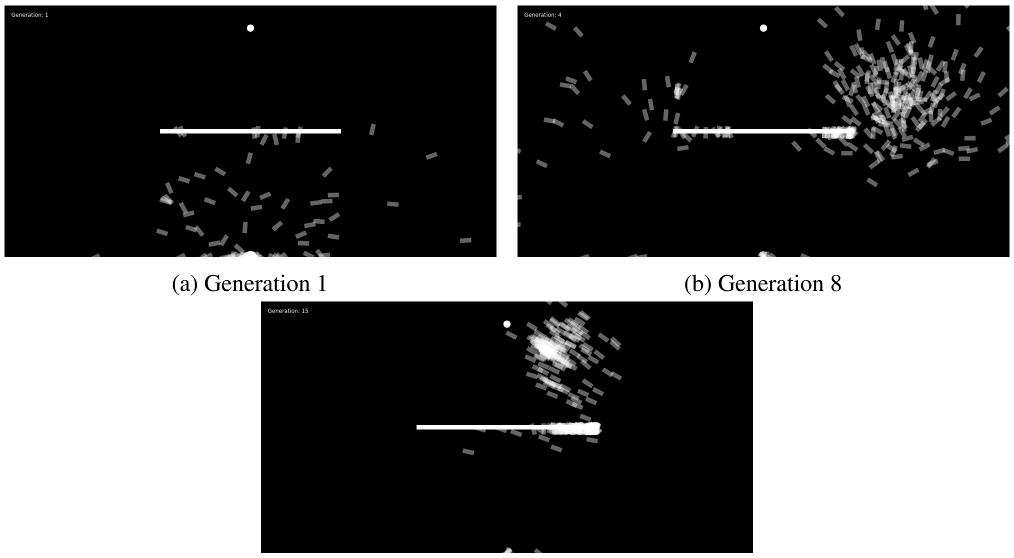
Reinforcement learning technics are useful for solving a wide variety of problems, including robot control, game playing and recommendation systems. They had been used to create some of the most advanced artificial intelligence software programs in the word, for instance, the DeepMind's AlphaGo system. AlphaGo is a computer program developed by DeepMind to play the famous Japanese board game Go. This is an abstract strategy board game for two players in which the aim is to surround more territory than the opponent. It is considered to be one of the most complex board games in the world. The machine used a combination of deep learning and Monte Carlo tree search to make its decisions. The deep learning component was used to analyze patterns in past games and predict the next move, while the Monte Carlo tree search component was used to evaluate the potential outcomes of each move and choose the one that was most likely to lead to a win. In October 2015, AlphaGo became the first artificial intelligence program to defeat a professional human player at the game of Go. This victory was considered a significant milestone in the field of artificial intelligence, as Go is a complex game with a very large number of possible moves, and the strategies used in the game are not fully understood even by top human players.

As a simple example, we have prepared a small program in Javascript where we use reinforcement learning to solve a problem. The agent is a moving particle with a policy that tells it how to move. Its goal is to arrive to the circle beyond the obstacle. When one runs the program, it initializes a population of many agents with random polices. Particles start wandering, after some time moving around a fitting function rewards the policies that brought the particles closer to the objective and punished those that made the particles hit the obstacle. The next generation starts with a mixture of the previous best polices and a few random mutations. Some kind of random perturbation in the previous policies, that is the mutations, are needed in order to have enough variability so that the particle can discover new behaviors. See figure 2 to visualize this generation process. The code is provided in our github repository.[1]

2 State of the art

The articles directly related to the topic under study are [3](Àlex and David), [8](Àlex and David) and [5](David). The first one, is the article on which we are most focused, since it explains in detail how they used deep learning and reinforcement learning to discover new mathematical algorithms. Article [8] is about AlphaZero, the algorithm on which paper [3] is based to build AlphaTensor. Finally, in [5] we review how new and old methods might complement each other. Three days after the paper [3], two Austrian researchers show that they used a conventional computer-aided search to further improve one of the algorithms that the neural network had discovered.

[1]`https://github.com/davidrosado4/Workshop-ML.git`



(a) Generation 1 (b) Generation 8

(c) Generation 16

Figure 2: Policy-based Reinforcement Learning example.

2.1 Motivation

After this brief introduction to reinforcement learning, we are ready to focus on the main topic of this paper, which is, discovering faster matrix multiplication algorithms with reinforcement learning. Let us change the course and focus on the task of multiplying matrices. It is well known the importance of this computation in applied sciences such as data science, in which a large amount of data is available, and many matrix multiplications are needed to apply certain methods. Without going further, a neural network is based on matrix multiplication. Remember that in the standard recipe to multiply two $n \times n$ matrices, the order of n^3 multiplications are needed, hence if n is huge, such as matrices in data science, the problem is computationally expensive. Researchers have used a variety of strategies to tackle this issue over the past 50 years, many of which were based on computer searches assisted by human intuition. On October 2022, in an article published in Nature [3], a team from DeepMind demonstrated how to approach the problem in a novel way. They reported that they had effectively trained a neural network to find new, quick methods for matrix multiplication.

But how can a neural network play a pivotal role here? The discovered algorithm is especially open to automation, since matrix multiplication algorithms can be described as low-rank decompositions of a particular three-dimensional tensor (we will see this in the detail in the following section). Nevertheless, finding low-rank decompositions of 3D tensor is NP-hard and is also hard in practice. Therefore, in [3], they use deep reinforcement learning to learn to recognize and generalize over patterns in tensors, and use the learned agent to predict efficient decompositions. They reformulate the matrix multiplication algorithm discovery into a single-player game called TensorGame. The player chooses how to combine different entries of the matrices to multiply at each step and a score is assigned based on the number of selected operations required to reach the correct multiplication result. To solve TensorGame, they develop a method that combines model-based reinforcement learning and deep neural networks called AlphaTensor, which is based on AlphaZero.

AlphaZero is a computer program developed by DeepMind that plays board games and other strategic games such as chess, shogi, and go. In just one day of training, AlphaZero achieved superhuman performance starting from random play and given no domain knowledge except the game rules [8]. The algorithm has defeated the world champion players in the games of chess (Stockfish), shogi (elmo) and Go (AlphaGo). The agent learned by self-play, that is, playing against itself.

2.2 Mathematical background

A piece of prior knowledge of tensors might be required to understand the main idea of DeepMind's approach to AlphaTensor. In mathematics, a tensor is an algebraic object that describes a multilinear relationship between sets of algebraic objects related to a vector space. It can be understood as a multidimensional array of data. It is a generalization of a matrix, which is a two-dimensional array, and can be thought of as a collection of multiple matrices arranged along multiple dimensions.

Let us define also the tensor product. In mathematics, the tensor product $A \otimes B$ of two vector spaces A and B is a vector space to which is associated a bilinear map

$$\begin{aligned} A \times B &\to A \otimes B \\ (a, b) &\mapsto a \otimes b \end{aligned}$$

It can be understood as a binary operation that produces a tensor from two tensors. It is a generalization of the matrix product and is used to combine tensors of different types, such as vectors and matrices.

We will not provide further mathematical details, since it is not the goal of this paper. Nevertheless, for more information, we recommend the reader to consult [1](Àlex and David) or [9](David).

3 Algorithm

The algorithm is based on the representation of the matrix multiplication into a 3D tensor. For instance, we can represent a 2×2 matrix multiplication in a 3D tensor of size $4 \times 4 \times 4$. Let us see how. Assume the following notation:

$$\begin{pmatrix} c_1 & c_2 \\ c_3 & c_4 \end{pmatrix} = \begin{pmatrix} a_1 & a_2 \\ a_3 & a_4 \end{pmatrix} \begin{pmatrix} b_1 & b_2 \\ b_3 & b_4 \end{pmatrix}$$

As $c_1 = a_1 b_1 + a_2 b_3$, the tensor entries located in (a_1, b_1, c_1) and (a_2, b_3, c_1) are set to 1. Merely, the tensor specifies which entries from the input matrices to read, and where to write the result. In general, we will call $\mathcal{T}_n$ the 3D tensor of size $n^2 \times n^2 \times n^2$ describing the $n \times n$ matrix multiplication, or $\mathcal{T}_{n,m,p}$ the tensor describing the $(n \times m)(m \times p)$ matrix multiplication.[2] The key idea is, the decomposition of $\mathcal{T}_n$ into R rank-one terms, since this provides an algorithm for multiplying arbitrary $n \times n$ matrices using R scalar multiplications. By a decomposition of $\mathcal{T}_n$ into R rank-one terms, we mean

$$\mathcal{T}_n = \sum_{i=1}^{R} \mathbf{u}^{(r)} \otimes \mathbf{v}^{(r)} \otimes \mathbf{w}^{(r)}$$

where $\mathbf{u}^{(r)}$, $\mathbf{v}^{(r)}$ and $\mathbf{w}^{(r)}$ are vectors.

The problem is to find this decomposition, more precisely, find the previous vectors. For the agent to learn to solve the problem one must define the environment and how can the agent act on it. As said before, this environment is called TensorGame and the agent can act by subtracting the tensor of rank-one triplets. The game is initialized with the target tensor we wish to decompose: $S_0 = \mathcal{T}_n$. Then, at t step, the game produces a tensor S_t in the following way: the player selects a triplet $(\mathbf{u}^{(t)}, \mathbf{v}^{(t)}, \mathbf{w}^{(t)})$ and the tensor S_t is updated by

$$S_t \leftarrow S_{t-1} - \mathbf{u}^{(t)} \otimes \mathbf{v}^{(t)} \otimes \mathbf{w}^{(t)}.$$

The goal of the player is to reach the zero tensor, i.e., $S_t = 0$ by applying the smallest number of moves. We limit the steps to a maximum value, R_{max}, to avoid playing unnecessarily long games. If for some $t < R_{max}$, the player reaches the zero tensor, it is guaranteed the correctness of the resulting matrix multiplication algorithm.

[2]Notice that $\mathcal{T}_n = \mathcal{T}_{n,n,n}$.

The rewards and penalties are as follows: for every step, a reward of -1 is given in order to find the zero tensor with the fewest number of steps. If the game ends without finding the zero tensor, after R_{max} steps, the agent receives an additional terminal reward $-\gamma(S_{R_{max}})$, that is an upper bound on the rank of the terminal tensor.

Let us continue explaining how AlphaTensor, plays this TensorGame. It uses a combination of a deep neural network and Monte Carlo tree search to make its decisions, as its predecessors AlphaGo and AlphaZero. Monte Carlo search tree (MCTS) is a best-first search method based on a randomized exploration of the search space. Then using the results of previous exploration, the algorithm becomes able to predict the most promising moves more accurately, and thus, their evaluation becomes more precise [2] (Àlex). The input for the neural network is a tensor, S_t, and the output is a policy and a value. The policy provides a distribution over potential actions, thus it relies on sampling actions instead of picking a specific one. The value provides an estimate of the cumulative reward starting from the current state S_t. Then it uses MCST from the given policy to choose the best next action. The algorithm starts from $\mathcal{T}_n$ and uses this procedure until it reaches the zero tensor. Finished games are then used as feedback for the network to tune the parameters.

One of the major challenges of TensorGame and other problems alike, is that the set of potential actions, $(\mathbf{u}^{(r)}, \mathbf{v}^{(r)}, \mathbf{w}^{(r)})$ in our case, at each step is enormous. Therefore, several technics specific for each kind of task should be studied and implemented. For instance, the neural network architecture consists of a transformer with an attention mechanism inspired by the invariance of the tensor rank to slice reordering. The output of the transformer is sent to an autoregressive model to obtain the policy distribution and to a multilayer perceptron to obtain the value. Other technics used to improve the efficiency of the algorithm are: synthetic demonstrations, which consists of sampling factors $(\mathbf{u}^{(r)}, \mathbf{v}^{(r)}, \mathbf{w}^{(r)})$ at random, constructing the tensor and train the network on a mixture of supervised loss and reinforcement learning loss; change of basis, since an efficient algorithm is invariant to change of basis, a random change of basis is made at the beginning to add diversity; and data augmentation, as factorizations are order invariant, we can build additional tensor factorization training pairs by swapping actions of each finished game.

4 Experiments and results

Trained from random, AlphaTensor is able to discover matrix multiplication algorithms that perform more efficiently than previously known human and computer-designed algorithms. One of the most important strengths of AlphaTensor is its capacity to deal with complex stochastic an non-differentiable rewards. By changing the fitting function to give a reward based on executing time, the model is able to find matrix multiplication algorithms that are efficient specifically for the kind of hardware it was trained on. Furthermore, it is flexible enough that it can work in a wide variety of spaces other than the real numbers, for instance finite fields.

A single AlphaTensor agent was trained to find matrix multiplication algorithms for matrix sizes $n \times m$ with $m \times p$, where $n, m, p \leq 5$. The discovered algorithms can be recursively applied to multiply matrices of arbitrary size. Also, this agent finds algorithms for two different arithmetic, which are modular arithmetic (over the finite field of modulo 2, $\mathbb{Z}/\mathbb{Z}_2$), and standard arithmetic (over the field of real numbers $\mathbb{R}$).

With these settings, after playing TensorGame, AlphaTensor re-discovers the best known algorithms for multiplying matrices and even improves the efficiency for several matrix sizes. The most interesting results are the improvement for sizes 4×4 in $\mathbb{Z}/\mathbb{Z}_2$, that uses 47 multiplications, outperforming the Strassen's algorithm which involves 49 multiplications. Also improves the algorithm for multiplying 4 with 5×5 in the reals, with 76 multiplications instead of 80. A large dataset with this and other results is available and free to use in their repository[3]. This matrix multiplication algorithms are in terms of the tensor parameterized by its one-rank decomposition $\{\mathbf{u}^{(r)}, \mathbf{v}^{(r)}, \mathbf{w}^{(r)}\}_{r=1}^{R}$, that is $\mathcal{T}_{n,m,p} = \sum_{r=1}^{R} \mathbf{u}^{(r)} \otimes \mathbf{v}^{(r)} \otimes \mathbf{w}^{(r)}$. Therefore, in order to compute the matrix multiplication $\mathbf{C} = \mathbf{AB}$, given the matrices A and B of sizes $n \times m$ and $m \times p$, respectively, one should use the following algorithm:

[3]`https://github.com/deepmind/alphatensor`

Algorithm for computing matrix multiplication.

Parameters: $\{\mathbf{u}^{(r)}, \mathbf{v}^{(r)}, \mathbf{w}^{(r)}\}_{r=1}^{R}$, such that $\mathcal{T}_{n,m,p} = \sum_{r=1}^{R} \mathbf{u}^{(r)} \otimes \mathbf{v}^{(r)} \otimes \mathbf{w}^{(r)}$.
Input: A and B matrices of sizes $n \times m$ and $m \times p$, respectively, reshaped as vectors $\mathbf{a}, \mathbf{b}$.
Output: $\mathbf{C} = \mathbf{AB}$

```
(1) for $r = 1, \dots, R$ do:
(2)     $m_r \longleftarrow (u_1^{(r)} a_1 + \cdots + u_{nm}^{(r)} a_{nm})(v_1^{(r)} b_1 + \cdots + v_{mp}^{(r)} b_{mp})$
(3) for $i = 1, \dots, np$ do:
(4)     $c_i \longleftarrow w_i^{(1)} m_1 + \cdots + w_i^{(R)} m_R$

return C
```

In our repository[4] one can find a Colab with an example of how to load the algorithms discovered by AlphaTensor and the implementation of the above algorithm. Furthermore, if there is a GPU available, one can use DeepMind's `benchmarking` folder, which contains a script that can be used to measure the actual speed of matrix multiplication algorithms on an NVIDIA V100 GPU. After many trials, we were not able to run the code because of incompatibilities with our GPU, but we encourage the reader to try it themself. To do so, just go to `https://github.com/deepmind/alphatensor/tree/main/benchmarking` and follow the installation guide, notice that a GPU compatible with NVIDIA V100 GPU must be available in your hardware.

5 Conclusions

We have analyzed a deep reinforcement learning technic that solve in a better manner a specific mathematical problem. However, AlphaTensor still have some limitations. The value that the coefficients of the factors of the tensor can take, need to be fixed. In order to avoid problems of precision with floating point numbers and in order to find algorithms that work well with any kind of coefficient field, values are set to be +1, -1 or 0 [4](Àlex). This can potentially lead to skipping efficient algorithms, that is why they are already considering a future research to adapt AlphaTensor to optimize also such coefficients. Also, there is the need to properly define an environment with rules that makes sense where the model can play against itself. That can be a difficult issue to overcome if we want to explore the algorithms for a mathematical problem that is hardly explainable in this terms. Nevertheless, twenty-first century is the era of the newborn artificial intelligence able to outperform human mastery for many different problems. We first saw that with AlphaGo, breaking the most challenging board game Go with strategies that some experts even considered 'original' or 'beautiful'. Then AlphaZero, was not only able to learn to play a game, but to surpass any human in many different video games. And the same reinforcement learning approach leads to AlphaTensor that finds solutions for mathematical problems much more efficient than known human algorithms. The latest DeepMind's model demonstrates the viability of deep reinforcement learning to address difficult mathematical problems, and potentially assisting mathematicians in discoveries.

References

[1] Michael Atiyah. *Introduction to commutative algebra*. CRC Press, 2018.

[2] Guillaume M J-B Chaslot, Mark H M Winands, H Jaap VAN DEN Herik, Jos W H M Uiterwijk, and Bruno Bouzy. Progressive strategies for Monte-Carlo tree search. *New Math. Nat. Comput.*, 04(03):343–357, 2008.

[3] Alhussein Fawzi, Matej Balog, Aja Huang, Thomas Hubert, Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Francisco J R Ruiz, Julian Schrittwieser, Grzegorz Swirszcz, et al. Discovering faster matrix multiplication algorithms with reinforcement learning. *Nature*, 610(7930):47–53, 2022.

[4] Marijn J. H. Heule, Manuel Kauers, and Martina Seidl. New ways to multiply 3 x 3-matrices. *CoRR*, abs/1905.10192, 2019.

[4]`https://github.com/davidrosado4/Workshop-ML/tree/main/alphatensor`

[5] Manuel Kauers and Jakob Moosbauer. The fbhhrbnrssshk-algorithm for multiplication in $\mathbb{Z}_2^{5\times 5}$ is still not the end of the story 2. *arXiv preprint arXiv:2210.04045*, 2022.

[6] Ofir Nachum, Mohammad Norouzi, Kelvin Xu, and Dale Schuurmans. Bridging the gap between value and policy based reinforcement learning. *Advances in neural information processing systems*, 30, 2017.

[7] Stuart Russell and Peter Norvig. Inteligencia artificial. un enfoque moderno. 2. a edición, 2004.

[8] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018.

[9] Oscar Zariski and Pierre Samuel. *Commutative algebra: Volume II*, volume 29. Springer Science & Business Media, 2013.