Better Matrix Multiplication Through Deep Reinforcement Learning

Frederik Hirsch

January 2024

Abstract

DeepMind's deep reinforcement learning (DRL) agent *AlphaTensor* aims at finding lower-rank matrix multiplication algorithms through formulating and playing tensor decomposition as a single-player game. *AlphaTensor* succeeded in finding new matrix multiplications algorithms that improved or matched current best known algorithms for smaller matrices up to sizes of 5×5. Experiments showed that it can also be utilized to optimize other properties of matrix multiplication. This report aims at giving the reader the background knowledge to understand how tensor decomposition works, as well as providing an introduction and overview of the DRL agent *AlphaTensor*.

1 Introduction

Matrix multiplication is one of the most important operations in computer science today. Machine learning, graphical applications and many more tasks of modern computers rely heavily on matrix multiplication as one of their fundamental building blocks [3]. Therefore it is of great interest to optimize it as much as possible, because every tiny improvement can result in large computational savings when it is performed millions of times a second.

At first it might seem that such a seemingly simple and well defined operation as matrix multiplication is something that should be understood well enough to find an optimal algorithm for computing it. The problem lies in finding a way to reduce the number of multiplications needed in the algorithm, also called rank. While the standard matrix multiplication algorithm always needs n^3 multiplications for multiplying two $n \times n$ matrices with each other, there are ways to do this with less as first discovered by Volker Strassen in 1969 [8]. He proposed an algorithm that multiplies two 2×2 matrices with just 7 multiplications, which was soon proven the minimum number of multiplications needed for 2×2 matrices [10].

Since then, finding more efficient matrix multiplication algorithms has been an ongoing field of research, but even for matrices as small 3×3 the minimum number of multiplications isn't known to this day [2]. Until now most research has been done by human search involving the solving of large equation systems [5], combinatorial approaches using SAT Solvers [4] or combining known algorithms for smaller matrices to improve the rank of matrix multiplication for larger matrices [6].

Googles subsidiary *DeepMind* brings a new approach to this field with *AlphaTensor*, which is based on *AlphaZero*, a deep reinforcement learning (DRL) agent that achieved superhuman performance in the games of Go, chess and shogi [7]. *AlphaTensor* uses DRL to find new matrix multiplication algorithms which use less multiplications. It does so by turning the search for new algorithms into a game, which can then be played by the agent in a similar way to the above mentioned games. This approach relies less on human guidance and is therefore expected to find more optimal solutions with less human bias [2].

But before taking a closer look at *AlphaTensor* itself, we first need to understand how a matrix multiplication algorithm can be formalized such that a computer can deal with it better. This will be explained in section 2. How this can be turned into a game, and how *AlphaTensor* can solve this game is explained in section 3. Finally, the results of *AlphaTensor* will be presented in section 4.

2 Formalizing Matrix Multiplication Algorithms

Preliminaries:

1. This section contains some definitions and examples involving matrices. For better readability, we refer to entries of the matrices not via 2 different indices (row, column) as it is normally done, but by only 1 index enumerating every entry row by row. For example, we refer to entries of a 2×2 matrix A the following way:

$$A = \begin{pmatrix} a_1 & a_2 \\ a_3 & a_4 \end{pmatrix}$$

2. For better understanding of the examples given, it is also helpful to have the standard matrix multiplication algorithm in mind, especially for multiplying 2×2 matrices $C = AB$:

$$c_1 = a_1b_1 + a_2b_3 \qquad c_2 = a_1b_2 + a_2b_4$$
$$c_3 = a_3b_1 + a_4b_3 \qquad c_4 = a_3b_2 + a_4b_4$$

3. The outer product $u \otimes v$, where u and v are vectors of length m and n is defined as:

$$u \otimes v := \begin{pmatrix} u_1v_1 & u_1v_2 & \dots & u_1v_n \\ u_2v_1 & u_2v_2 & \dots & u_2v_n \\ \vdots & \vdots & \ddots & \vdots \\ u_mv_1 & u_mv_2 & \dots & u_mv_n \end{pmatrix}$$

   Later on, the 3-dimensional version of the outer product $u \otimes v \otimes w$ is used, with an additional vector w of length p. In that case, the result is a tensor (2.1), where p matrices are layered behind one another, and the matrix in layer k is defined as $w_k \cdot [u \otimes v]$.

2.1 Matrix Multiplication as a Tensor

To understand how *AlphaTensor* finds new ways to multiply matrices, we first need to understand what tensors are and how matrix multiplication relates to them. A tensor can be defined as a multidimensional array of numbers. For example, a vector is a tensor of dimension 1 and a matrix is a tensor of dimension 2.

Because matrix multiplication is a bilinear function (a function that is linear in both of its arguments) between vector spaces, it can be represented as a 3 dimensional tensor, similarly to how any linear function between vector spaces can be represented as a matrix[2]. The size of this tensor is dependent on the size of the matrices that are multiplied. For matrix multiplication with square matrices of size n, the matrix multiplication tensor $\mathcal{T}_n$ has dimension (n^2, n^2, n^2). This is because the vector space $\mathcal{M}^{n \times n}$ consisting of the square matrices of size n has dimension n^2, and matrix multiplication (f_n) in this vector space has type $f_n : \mathcal{M}^{n \times n} \times \mathcal{M}^{n \times n} \to \mathcal{M}^{n \times n}$. More general, for matrix multiplication of $m \times n$ and $n \times p$ matrices, the function has type $f_{m,n,p} : \mathcal{M}^{m \times n} \times \mathcal{M}^{n \times p} \to \mathcal{M}^{m \times p}$, therefore $\mathcal{T}_{m,n,p}$ has dimension $(m \times n, n \times p, m \times p)$ [2].

The values of $\mathcal{T}$ are independent of the specific matrices that are multiplied, as they just define which entries of matrix A to multiply with which entries of matrix B to get matrix $C = AB$. To be more specific, the function $f_{m,n,p}$ is given by $\mathcal{T}_{m,n,p}$ through the following definition, where $t_{i,j,k}$ refers to the entry of $\mathcal{T}_{m,n,p}$ in row i, column j and layer k:

$$f_{m,n,p} : \mathcal{M}^{m \times n} \times \mathcal{M}^{n \times p} \to \mathcal{M}^{m \times p}$$
$$(A, B) \mapsto C, \text{ with } c_k = \sum_{\substack{1 \le i \le m \cdot n \\ 1 \le j \le n \cdot p}} t_{i,j,k} \cdot a_i \cdot b_j$$

This means that the entries of $\mathcal{T}$ must always be 0 or 1, depending on whether the product $a_i \cdot b_j$ is needed for the calculation of c_k. For a better understanding of how matrix multiplication tensors look, $\mathcal{T}_2$ is shown in figure 1.

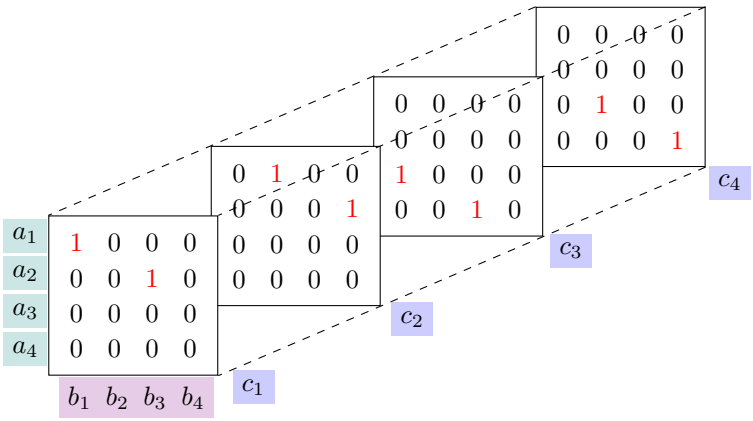

Figure 1: *Tensor $\mathcal{T}_2$ for multiplication of square matrices of size 2. [2]*

2.2 Tensor Decomposition

Having matrix multiplication as a tensor alone doesn't directly yield a faster way to multiply matrices. For that, we need to look at decompositions of that tensor. In general, a decomposition of a tensor is a series of elementary operations, such as addition and multiplication, on a set of (mostly simpler) tensors, that result in the original tensor.

In our case, we are searching for a decomposition into a sum of rank-one tensors, where the number of rank-one tensors is as small as possible. A rank-one tensor is a tensor which can be expressed as an outer product of vectors: $\mathcal{X}_{rank\text{-}one} = u \otimes v \otimes w$. A decomposition of a matrix multiplication tensor $\mathcal{T}_n$ into R rank-one terms can then be defined as

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

where $u^{(r)}$, $v^{(r)}$ and $w^{(r)}$ are all vectors of length n^2 [2]. When a decomposition consists of R rank-one terms, it has rank R.

Each rank-one term of a decomposition corresponds to one multiplication in the matrix multiplication algorithm. For example, the rank-one term defined by $u = (0,1,0,1)^T$, $v = (1,0,-1,0)^T$ and $w = (1,1,0,0)^T$ denotes that the term $(a_2 + a_4) \cdot (b_1 - b_3)$ is used for the calculation of c_1 and c_2. For square matrices of size 2, the standard way of multiplying matrices is a tensor decomposition with 8 rank-one terms, as there are 8 different product terms used.

One of the most popular multiplication algorithms that improves on the number of terms is the Strassen-Algorithm [8] which only uses 7 terms for multiplying 2×2 matrices. It can also be utilised for multiplying larger matrices by recursive application of the same algorithm. The decomposition of the Strassen-Algorithm is shown in figure 2.

$$\begin{aligned}
m_1 &= (a_1 + a_4) \cdot (b_1 + b_4) \\
m_2 &= (a_3 + a_4) \cdot b_1 \\
m_3 &= a_1 \cdot (b_2 - b_4) \\
m_4 &= a_4 \cdot (b_3 - b_1) \\
m_5 &= (a_1 + a_2) \cdot b_4 \\
m_6 &= (a_3 - a_1) \cdot (b_1 + b_2) \\
m_7 &= (a_2 - a_4) \cdot (b_3 + b_4)
\end{aligned}$$

$$\begin{aligned}
c_1 &= m_1 + m_4 - m_5 + m_7 \\
c_2 &= m_3 + m_5 \\
c_3 &= m_2 + m_4 \\
c_4 &= m_1 - m_2 + m_3 + m_6
\end{aligned}$$

Figure 2: *Strassen Algorithm for multiplying 2×2 matrices.* [8]

3 Finding New Tensor Decompositions

For reinforcement learning (RL), two main components are needed: an environment and an agent that interacts with the environment. How these two work for finding new matrix multiplications is explained in the following two sections.

3.1 Tensor Decomposition as a Game

To be able to apply RL to the problem, we need to turn tensor decomposition into a game with different states, actions that transform states into other states, as well as an end goal. *DeepMind* called this *TensorGame* [2], and it works as follows:

- **Game State:** The state of the game at each step t is defined by a tensor $\mathcal{S}_t$. At the beginning of the game, $\mathcal{S}_0$ is set to $\mathcal{T}_{m,n,p}$, the tensor that is to be decomposed. The game state tensor always has the dimensions of the tensor that is decomposed in the game.
- **Actions:** An action at each step t is defined by three vectors $u^{(t)}$, $v^{(t)}$ and $w^{(t)}$, which the player chooses. The new game state is then calculated by subtracting the resulting rank-one tensor from the previous game state: $\mathcal{S}_t \leftarrow \mathcal{S}.t-1-u^{(t)} \otimes v^{(t)} \otimes w^{(t)}$. The scalar entries of u, v and w are limited to a discrete set of values such as $\{-2,-1,0,1,2\}$, because using floating point numbers would in practice lead to inaccurate algorithms due to the finite precision of floating point arithmetic.
- **Goal:** The goal of the game is to reach the zero tensor $\mathcal{S}_t = 0$, where all entries are zero. When adding up all the actions leading to the zero tensor, we then get $\mathcal{T}_{m,n,p} = \sum_{i=1}^{t} u^{(i)} \otimes v^{(i)} \otimes w^{(i)}$, which means that we have found a valid decomposition. The number of steps in the game can be limited to an already known upper bound of the rank of $\mathcal{T}_{m,n,p}$ (R_{limit}), because reaching the goal in more steps would only result in a decomposition with more terms.
- **Reward Function:** To guide the RL agent in finding a decomposition with a rank as low as possible, the reward function gives a negative reward of -1 with each step taken, as well as an additional reward $-\gamma(\mathcal{S}_{R_{limit}})$ at the end of the game, where $\gamma(\mathcal{S}_{R_{limit}})$ is an upper bound of the rank of the tensor $\mathcal{S}_{R_{limit}}$ that is left when the step limit is reached. This means that when the zero tensor is reached at the end of the game, the additional reward is 0.

3.2 Exploring *TensorGame* with Reinforcement Learning

Additionally to the environment (which is given by *TensorGame*), the other key part of RL is the agent that interacts with the environment and develops a policy π (a function that returns the optimal actions based on the given state) based on the rewards it gets from the environment [1].

The biggest challenge for the agent that interacts with *TensorGame* is the extremely large action space. While in games like Chess and Go there are up to a few 100 actions that can be taken in each state, the size of the action space for *TensorGame* is over 10^{12} in most relevant cases [2]. For example, when decomposing $\mathcal{T}_3$, the agent needs to choose 3 vectors of length 9, with 5 possibilities (3.1) for each entry of the vectors. This results in $5^{3 \cdot 9} > 10^{18}$ different actions to choose from.

To stand up to this challenge, *DeepMind* proposes their DRL agent *AlphaTensor*, which is based on *AlphaZero*, a more generalized successor of their superhuman Go-program *AlphaGo*. Explaining the full architecture would be beyond the scope of this paper, so I will just give a short informal overview and explain some interesting techniques used for training the agent. For a more detailed and precise explanation of *AlphaTensor* and the machine learning principles behind it, I refer to the following papers: [2, 7, 9].

The main component of *AlphaTensor* is a transformer based [9] deep neural network, which produces a policy π as well as a value function z (a probability distribution over the sums of the future rewards) from the current state $\mathcal{S}_t$ and previous states of the game [2]. These two functions are the critical part in finding the best action to take in a given state. The agent uses Monte Carlo tree search (MCTS) to find the best path in the search tree, where nodes represent different states of the game (with the current state as root node) and edges actions that can be taken from each state. MCTS is a sample based algorithm, which uses the policy π to sample different actions that lead to high value states (determined by z). When the end of a game is reached on a path, the result is used to train the deep neural network, so that it can put out more accurate policies and value functions in the future [2].

The neural network of *AlphaTensor* is trained on pairs of tensors, and their decompositions. To achieve better results, multiple different techniques are used to generate additional training data from the self played games as well as synthetic data from scratch:

- **Synthetic training data**: While it is computationally very difficult to find decompositions for a given tensor, it is very easy to build a tensor from any given decomposition. To generate pairs of tensors and their decompositions from scratch, multiple pairs of vectors u, v and w are sampled which represent the decomposition, which are then multiplied and added together to get the tensor. Theses pairs can then be used as training data for the neural network [2].
- **Change of basis**: So far we only looked at the matrix multiplication tensor in the canonical basis. The rank of a tensor is actually independent of the basis used, and any decomposition found for a tensor in a different basis can easily be transformed back into the canonical basis. That means that any found decomposition can be transformed to a different basis to receive a new (tensor, decompositon) pair to train the neural network. Additionally, a tensor can be transformed to multiple different bases, to search for decompositions in all of them [2].
- **Permutations of decompositions**: As the different terms of the decomposition are just added up, the order of them doesn't matter for the result. Because of this, it is possible to just change the order in found decompositions to receive a new training pair [2].

All these components together, namely MCTS combined with a powerful deep neural network trained on self-played and synthetic data allow *AlphaTensor* to handle the extremely large action space of tensor decomposition.

4 Results

A single *AlphaTensor* agent was trained to find algorithms for multiplication of matrices of sizes $m \times n$ with $n \times p$, where $2 \leq m, n, p \leq 5$. For all of these sizes it found decompositions which either have the same rank as the previous best known decompositions, or improved the rank [2]. It not only searched for multiplication algorithms for matrices over $\mathbb{R}$, but also for algorithms in modular arithmetic, namely for matrices over the quotient ring $\mathbb{Z}_2$ (which means that the matrices can only have entries 0 and 1). The improvements *AlphaTensor* made over the previous best known ranks is shown in figure 3.

Size (m, n, p)	Best rank known	AlphaTensor rank ($\mathbb{Z}_2$)	AlphaTensor rank ($\mathbb{R}$)
(4, 4, 4)	49	47	49
(5, 5, 5)	98	96	98
(3, 4, 5)	48	47	47
(4, 4, 5)	64	63	63
(4, 5, 5)	80	76	76

Figure 3: *Comparison of the best known rank for matrix multiplication before AlphaTensor (column 2) and the best rank found by AlphaTensor for modular arithmetic (column 3) and standard arithmetic (column 4). Sizes are chosen to only show the ones where AlphaTensor found an improved rank. For all other sizes up to* $(5, 5, 5)$ *it matched the rank, but didn't improve upon it.* **This table is a shortened version of Fig. 3 in the *AlphaTensor* paper [2].**

Despite only directly searching for algorithms up to matrix size $5, 5, 5$, *AlphaTensor* also improved the best rank for many larger matrix multiplications with $n, m, p \leq 12$ by combining known algorithms for smaller matrices recursively. For example, the rank for multiplying matrices $(6, 8, 10)$ has been improved from $336 = 7 \cdot 48$ to $329 = 7 \cdot 47$, by combining the algorithms for $(3, 4, 5)$ (rank 47, improved from 48) and $(2, 2, 2)$ (rank 7) [2]. For a table containing every rank improved via factorization see *Extended Data Table 1* in the *AlphaTensor* paper [2].

The only thing defining that *AlphaTensor* searches for decompositions with the lowest rank possible is the reward function. By changing the reward function, it can optimise other properties such as practical efficiency on specific hardware. *DeepMind* used *AlphaTensor* to optimize the multiplication of 8192×8192 square matrices on the Nvidia V100 GPU as well as their own Tensor

Processing Unit (TPU) by searching for the 4×4 algorithm which results in the fastest benchmark when combined with the already very efficient implementation for 2048×2048 multiplication. For that, they added a benchmark ran on the specific hardware to the reward function whenever a valid solution was found. The agent doesn't need to know anything about the architecture of the hardware, but just sees it as a blackbox that it gets a benchmark score from. For the Nvidia V100 GPU *DeepMind* reported a speed-up of 8.5% using *AlphaTensor*, compared to a speed-up of 4.3% using the recursive Strassen algorithm for 4×4 multiplication. The speed-ups are both measured relative to the same standard. For the TPU they reported speed-ups of 10.3% with *AlphaTensr* and 6.6% with the Strassen method [2].

These results show that DRL is useful for helping to solve complex mathematical problems, as well as bridging the gap between theoretical optimization of an algorithm and a practical improvement in performance [3]. *DeepMind* is confident that the flexibility given by *AlphaTensor* can be utilized to find algorithms optimizing other metrics such as numerical stability or energy usage [2].

References

[1] Kai Arulkumaran, Marc Peter Deisenroth, Miles Brundage, and Anil Anthony Bharath. "Deep Reinforcement Learning: A Brief Survey". In: *IEEE Signal Processing Magazine* 34.6 (2017), pp. 26–38.

[2] 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". In: *Nature* 610.7930 (2022), pp. 47–53.

[3] Samuel Greengard. "Better Algorithms through Faster Math". In: *Communications of the ACM* 66.6 (2023), pp. 11–13.

[4] Marijn J.H. Heule, Manuel Kauers, and Martina Seidl. "New ways to multiply 3×3-matrices". In: *Journal of Symbolic Computation* 104 (2021), pp. 899–916.

[5] Julian D. Laderman. "A noncommutative algorithm for multiplying 3 x 3 matrices using 23 multiplications". In: *Bulletin of the American Mathematical Society* 66.1 (1976), pp. 126–128.

[6] Alexandre Sedoglavic and Alexey V. Smirnov. "The Tensor Rank of 5x5 Matrices Multiplication is Bounded by 98 and its Border Rank by 89". In: *Proceedings of the 2021 on International Symposium on Symbolic and Algebraic Computation*. ISSAC '21. Association for Computing Machinery, 2021, pp. 345–351.

[7] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. "A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play". In: *Science* 362.6419 (2018), pp. 1140–1144.

[8] Volker Strassen et al. "Gaussian elimination is not optimal". In: *Numerische mathematik* 13.4 (1969), pp. 354–356.

[9] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. "Attention is All you Need". In: *Advances in Neural Information Processing Systems*. Ed. by I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett. Vol. 30. Curran Associates, Inc., 2017.

[10] S. Winograd. "On multiplication of 2 × 2 matrices". In: *Linear Algebra and its Applications* 4.4 (1971), pp. 381–388.