Discovering faster matrix multiplication algorithms with reinforcement learning

Learning a modern method for the calculation of matrix multiplication

Eddie Sung
dept. mathematics of National Taiwan University
b09201029@ntu.edu.tw

***Abstract*—This paper gives a introduction and some operations of the article "Discovering faster matrix multiplication algorithms with reinforcement learning". The main concept of AlphaTensor and the result of our research will be presented in this paper.**

I. Introduction

Matrix multiplication is a fundamental operation in various scientific and engineering disciplines, playing a crucial role in fields such as computer graphics, machine learning, and numerical simulations. Despite its widespread use, the computational complexity of matrix multiplication remains a significant challenge, particularly as the size of the matrices involved increases. The classical algorithm for matrix multiplication has a time complexity of $O(n^3)$, which can become a bottleneck in many applications.

Recent advancements in artificial intelligence and machine learning have opened new avenues for optimizing computational tasks. In particular, DeepMind's AlphaTensor, based on the principles of deep reinforcement learning (DRL), has demonstrated promising results in discovering more efficient algorithms for matrix multiplication. This report aims to explore the core concepts and methodologies presented in the paper "Discovering Faster Matrix Multiplication Algorithms with Reinforcement Learning" (2022), replicate its findings, and analyze the potential improvements and future directions in this research area.

We begin with an overview of the basics of matrix multiplication, followed by a detailed discussion of AlphaTensor, the tensor game, and the application of DRL in optimizing matrix multiplication. Comparisons between traditional algorithms and AlphaTensor's performance will be presented, along with an analysis of the results and potential future work.

II. Basics of Matrix Multiplication

Matrix multiplication is an operation that takes two matrices as input and produces a third matrix as output. If A is an $m \times n$ matrix and B is an $n \times p$ matrix, their product $C = AB$ is an $m \times p$ matrix. The element c_{ij} of matrix C is computed as:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj} \tag{1}$$

where a_{ik} is the element in the i-th row and k-th column of matrix A, and b_{kj} is the element in the k-th row and j-th column of matrix B. This operation involves performing n multiplications and $n-1$ additions for each element of the resulting matrix C, leading to a total of $O(mnp)$ operations.

A. Computational Complexity

The naive algorithm for matrix multiplication, as described above, has a time complexity of $O(n^3)$ when $m = n = p$. This cubic complexity can become prohibitive for large matrices, prompting the development of more efficient algorithms. Notable improvements include Strassen's algorithm, which reduces the time complexity to approximately $O(n^{2.81})$, and the Coppersmith-Winograd algorithm, which further reduces it to $O(n^{2.376})$. However, these algorithms are often complex and challenging to implement.

B. Importance in Applications

Matrix multiplication is a cornerstone operation in various applications:

- **Computer Graphics:** Transformations and projections of 3D models.
- **Machine Learning:** Operations in neural networks, particularly in training phases involving large datasets.
- **Numerical Simulations:** Solving systems of linear equations, performing eigendecomposition, and other matrix-related computations.

Given its widespread application and the computational demands of modern data-intensive tasks, finding more efficient algorithms for matrix multiplication is of significant interest. This is where AlphaTensor's approach, leveraging deep reinforcement learning to discover new algorithms, presents a groundbreaking development.

In the following sections, we will delve into the specifics of AlphaTensor, the tensor game it uses for optimization, and the application of deep reinforcement learning in this context.

III. AlphaTensor

A. Introduction to AlphaTensor

AlphaTensor, an advanced AI model developed by DeepMind, specializes in optimizing matrix multiplication algorithms. Using cutting-edge machine learning methods, particularly deep reinforcement learning, AlphaTensor aims to find algorithms that can perform matrix multiplications with minimal computational steps. This innovation is crucial because

matrix multiplication is fundamental in many scientific and engineering domains. AlphaTensor builds upon the success of previous AI achievements like AlphaGo, which defeated human champions in Go, and the utilization of computer programs to solve complex mathematical problems such as the Four Color Problem.

B. Concept of Tensor

Tensors are mathematical constructs that generalize scalars, vectors, and matrices to higher dimensions. They represent multi-dimensional arrays of numerical values, playing vital roles in physics, engineering, and computer science. In the context of matrix multiplication, tensors provide a compact representation of the operation across multiple dimensions. For instance, a three-dimensional tensor captures the essence of matrix multiplication, with dimensions corresponding to input and output matrices indices.

C. Algorithms as Tensor Decomposition

Matrix multiplication, denoted as $C = AB$, can be represented by a three-dimensional tensor T. This tensor can be decomposed into a sum of rank-one tensors, which are outer products of vectors. The decomposition aims to minimize the rank R of T, thereby reducing the computational complexity of matrix multiplication.

The decomposition equation is as follows:

$$T_{ijk} = \sum_{r=1}^{R} u_i^{(r)} \otimes v_j^{(r)} \otimes w_k^{(r)}, \tag{2}$$

where $\otimes$ denotes the tensor product, and u, v, w are vectors. Discovering efficient low-rank decompositions is central to AlphaTensor's approach, leading to more effective matrix multiplication algorithms. Crucially, Algorithm 1 can be used to multiply block matrices. By using this algorithm recursively, one can multiply matrices of arbitrary size, with the rank R controlling the asymptotic complexity of the algorithm. In particular, $N \times N$ matrices can be multiplied with asymptotic complexity $O(N \log n(R))$; see ref.[2] for more details.

IV. TensorGame

The Tensor Game is played as follows(see Fig.1):

1) **Game Setup:** The start position S_0 of the game corresponds to the tensor T, representing the bilinear operation of interest, expressed in some basis.
2) **Game Process:**
   - In each step t of the game, the player writes down three vectors $(u^{(t)}, v^{(t)}, w^{(t)})$, which specify the rank-1 tensor $u^{(t)} \otimes v^{(t)} \otimes w^{(t)}$.
   - The state of the game is updated by subtracting the newly written down factor:

$$S_t \leftarrow S_t - u^{(t)} \otimes v^{(t)} \otimes w^{(t)} \tag{3}$$

   - The game continues until the state reaches the zero tensor $S_R = 0$. This indicates that the factors written down throughout the game form a factorization of the start tensor S_0, i.e.,

$$S_0 = \sum_{t=1}^{R} u^{(t)} \otimes v^{(t)} \otimes w^{(t)} \tag{4}$$

3) **Scoring:** The factorization obtained is then scored. For instance, when optimizing for asymptotic time complexity, the score is $-R$. When optimizing for practical runtime, the algorithm corresponding to the factorization $\{(u^{(t)}, v^{(t)}, w^{(t)})\}_{t=1}^{R}$ is constructed and then benchmarked on the fly.
4) **Game Limitation:** A limit R_{limit} is imposed on the maximum number of moves in the game to prevent unnecessarily long games. If the game ends due to reaching the move limit, a penalty score is given. This ensures that it is never advantageous to deliberately exhaust the move limit. For instance, when optimizing for asymptotic time complexity, this penalty is derived from an upper bound on the tensor rank of the final residual tensor $R_{\text{limit}}S$. This upper bound is obtained by summing the matrix ranks of the slices of the tensor.
5) **TensorGame over rings:** The decomposition of T^n is said to be in a ring E (defining the arithmetic operations) if each of the factors $u^{(t)}, v^{(t)}$, and $w^{(t)}$ has entries belonging to the set E, and additions and multiplications are interpreted according to E. The tensor rank depends, in general, on the ring. At each step of the Tensor Game, the additions and multiplications in equation (3) are interpreted in E. For example, when working in $\mathbb{Z}_2$ (in this case, the factors $u^{(t)}, v^{(t)}$, and $w^{(t)}$ live in $F = \{0, 1\}$), a modulo 2 operation is applied after each state update.
   Note that integer-valued decompositions $u^{(t)}, v^{(t)}$, and $w^{(t)}$ lead to decompositions in arbitrary rings E. Hence, provided F only contains integers, algorithms found in standard arithmetic apply more generally to any ring.

V. Deep Reinforcement Learning(DRL)

In this section, we delve into various aspects of Deep Reinforcement Learning (DRL), focusing on its application in solving complex problems. We start by exploring the Sample-based Monte Carlo Tree Search (MCTS) algorithm, a fundamental technique in DRL that leverages simulated trajectories to make decisions. Next, we discuss Synthetic Demonstrations, a method to enrich the agent's training data with pre-generated samples. We then examine the Change of Basis approach, which enhances exploration and exploits problem properties by transforming the input space. Additionally, we introduce Signed Permutations as a data augmentation technique and Action Canonicalization to ensure efficient policy learning. Finally, we provide insights into the Neural Network architecture used in DRL, highlighting its components and functionality. Overall, these topics form the foundation of our discussion on DRL methodologies and techniques.(see Fig.1)

A. Sample-based MCTS search

The sample-based MCTS search is very similar to the one described in Sampled AlphaZero. Specifically, the search consists of a series of simulated trajectories of TensorGame that are aggregated in a tree. The search tree therefore consists of nodes representing states and edges representing actions. Each state-action pair (s, a) stores a set of statistics $N(s, a)$, $Q(s, a)$, $\hat{\pi}(s, a)$, where $N(s, a)$ is the visit count, $Q(s, a)$ is the action value and $\hat{\pi}(s, a)$ is the empirical policy probability. Each simulation traverses the tree from the root state s_0 until a leaf state s_L is reached by recursively selecting in each state s an action a that has not been frequently explored, has high empirical policy probability and high value. Concretely, actions within the tree are selected by maximizing over the probabilistic upper confidence tree bound see ref.[3][4]:

$$\arg\max_a \left(Q(s,a) + c(s) \frac{\hat{\pi}(s,a)}{1 + N(s,a)} \right),$$

where $c(s)$ is an exploration factor controlling the influence of the empirical policy $\hat{\pi}(s, a)$ relative to the values $Q(s, a)$ as nodes are visited more often. In addition, a transposition table is used to recombine different action sequences if they reach the exact same tensor. This can happen particularly often in TensorGame as actions are commutative. Finally, when a leaf state s_L is reached, it is evaluated by the neural network, which returns K actions $\{a_i\}$ sampled from $\pi(a|s_L)$, alongside the empirical distribution $\hat{\pi}(a|s_L) = \sum_{i=1}^{K} \delta_{La_i}$. Differently from AlphaZero and Sampled AlphaZero, we chose v not to be the mean of the distribution of returns $z(\cdot|s_L)$ as is usual in most reinforcement learning agents, but instead to be a risk-seeking value, leveraging the facts that TensorGame is a deterministic environment and that we are primarily interested in finding the best trajectory possible. The visit counts and values on the simulated trajectory are then updated in a backward pass as in Sampled AlphaZero.

B. Synthetic Demonstrations

The synthetic demonstrations buffer contains tensor-factorization pairs, where the factorizations $\{(u_r, v_r, w_r)\}_{r=1}^{R}$ are first generated at random, after which the tensor

$$D = \sum_{r=1}^{R} u_r \otimes v_r \otimes w_r$$

is formed. We create a dataset containing 5 million such tensor-factorization pairs. Each element in the factors is sampled independently and identically distributed (i.i.d.) from a given categorical distribution over F (all possible values that can be taken). We discarded instances whose decompositions were clearly suboptimal (contained a factor with $u = 0$, $v = 0$, or $w = 0$).

In addition to these synthetic demonstrations, we further add to the demonstration buffer previous games that have achieved large scores to reinforce the good moves made by the agent in these games.

C. Change of Basis

The rank of a bilinear operation does not depend on the basis in which the tensor representing it is expressed, and for any invertible matrices A, B, and C, we have $\text{Rank}(T) = \text{Rank}(T(A, B, C))$, where $T(A, B, C)$ is the tensor after change of basis given by

$$T_{ijk} = \sum_{a=1}^{S} \sum_{b=1}^{S} \sum_{c=1}^{S} a_{ia} b_{jb} c_{kc} abc \tag{5}$$

Hence, exhibiting a rank-R decomposition of the matrix multiplication tensor T_n expressed in any basis proves that the product of two $n \times n$ matrices can be computed using R scalar multiplications. Moreover, it is straightforward to convert such a rank-R decomposition into a rank-R decomposition in the canonical basis, thus yielding a practical algorithm.

This approach has three appealing properties: (1) it provides a natural exploration mechanism as playing games in different bases automatically injects diversity into the games played by the agent; (2) it exploits properties of the problem as the agent need not succeed in all bases—it is sufficient to find a low-rank decomposition in any of the bases; (3) it enlarges coverage of the algorithm space because a decomposition with entries in a finite set $F = \{-2, -1, 0, 1, 2\}$ found in a different basis need not have entries in the same set when converted back into the canonical basis.

In full generality, a basis change for a 3D tensor of size $S \times S \times S$ is specified by three invertible $S \times S$ matrices A, B, and C. However, in our procedure, we sample bases at random and impose two restrictions: (1) $A = B = C$, as this performed better in early experiments, and (2) unimodularity ($\det(A) \in \{-1, +1\}$), which ensures that after converting an integral factorization into the canonical basis it still contains integer entries only (this is for representational convenience and numerical stability of the resulting algorithm).

D. Signed Permutations

In addition to playing (and training on) games in different bases, we also utilize a data augmentation mechanism whenever the neural network is queried in a new MCTS node. At acting time, when the network is queried, we transform the input tensor by applying a change of basis—where the change of basis matrix is set to a random signed permutation. We then query the network on this transformed input tensor, and finally invert the transformation in the network's policy predictions. Although this data augmentation procedure can be applied with any generic change of basis matrix (that is, it is not restricted to signed permutation matrices), we use signed permutations mainly for computational efficiency. At training time, whenever the neural network is trained on an (input, policy targets, value target) triplet, we apply a randomly chosen signed permutation to both the input and the policy targets, and train the network on this transformed triplet. In practice, we sample 100 signed permutations at the beginning of an experiment, and use them thereafter.

E. Action Canonicalization

For any λ_1, λ_2, $\lambda_3 \in \{-1, +1\}$ such that $\lambda_1\lambda_2\lambda_3 = 1$, the actions $(\lambda_1 u, \lambda_2 v, \lambda_3 w)$ and (u, v, w) are equivalent because they lead to the same rank-one tensor $(\lambda_1 u)\otimes(\lambda_2 v)\otimes(\lambda_3 w) = u \otimes v \otimes w$. To prevent the network from wasting capacity on predicting multiple equivalent actions, during training we always present targets (u, v, w) for the policy head in a canonical form, defined as having the first non-zero element of u and the first non-zero element of v strictly positive. This is well defined because u or v cannot be all zeros (if they are to be part of a minimal rank decomposition), and for any (u, v, w) there are unique λ_1, λ_2, $\lambda_3 \in \{-1, +1\}$ (with $\lambda_1\lambda_2\lambda_3 = 1$) that transform it into canonical form. In case the network predicts multiple equivalent actions anyway, we merge them together (summing their empirical policy probabilities) before inserting them into the MCTS tree.

F. Neural Network

The neural network architecture comprises a torso, policy head, and value head. The torso, based on modified transformers see ref.[5], operates over three $S \times S$ grids projected from the $S \times S \times S$ input tensors, enriching feature vectors with scalar information before projecting them into a 512-dimensional space. Attention-based blocks propagate information between the grids, facilitating representation learning. The policy head employs a transformer to model an autoregressive policy, while the value head predicts return distributions using a multilayer perceptron. This comprehensive architecture enables effective learning and decision-making in the tensor game environment.(see Fig.2)

Input: The input to the network contains all the relevant information of the current state and is composed of a list of tensors and a list of scalars. The most important piece of information is the current 3D tensor S_t of size $S \times S \times S$. In addition, the model is given access to the last h actions (usually set to 7), represented as h rank-1 tensors concatenated to the input. The list of scalars includes the time index t of the current action ($0 \leq t < R_{\text{limit}}$).

Torso: The torso of the network maps both scalars and tensors from the input to a representation useful for both policy and value heads. Its architecture, based on transformers, operates over three $S \times S$ grids projected from the $S \times S \times S$ input tensors. Each grid represents two out of the three modes of the tensor. The rest of the torso is a sequence of attention-based blocks with the objective of propagating information between the three grids. The representation sent to the policy head corresponds to the $3S^2$ 512-dimensional feature vectors produced by the last layer of the torso.

Policy Head: The policy head uses the transformer architecture see ref.[5] to model an autoregressive policy. Factors are decomposed into k tokens of dimensionality d such that $k \times d = 3S$. The transformer conditions on the tokens already generated and cross-attends to the features produced by the torso. At training time, teacher-forcing is used, where the ground truth actions are decomposed into tokens and taken as inputs into the causal transformer(see Fig.3).

Fig. 1. **Overview of AlphaTensor**. The neural network (bottom box) takes as input a tensor S_t, and outputs samples (u, v, w) from a distribution over potential next actions to play, and an estimate of the future returns (for example, of $-Rank(S_t)$). The network is trained on two data sources: previously played games and synthetic demonstrations. The updated network is sent to the actors (top box), where it is used by the MCTS planner to generate new games.

Fig. 2. **AlphaTensor's network architecture**. The network takes as input the list of tensors containing the current state and previous history of actions, and a list of scalars, such as the time index of the current action. It produces two kinds of outputs: one representing the value, and the other inducing a distribution over the action space from which we can sample from. The architecture of the network is accordingly designed to have a common torso, and two heads, the value and the policy heads. c is set to 512 in all experiments.

Value Head: The value head is composed of a multilayer perceptron whose last layer produces q outputs corresponding to the quantiles see ref.[6]. In this way, the value head predicts the distribution of returns from this state. At inference time, the agent is encouraged to be risk-seeking by using the average of the predicted values for quantiles over 75%(see Fig.3).

VI. Results and Analysis

In this experiment, we calculated the multiplication of two 2x2 matrices and two 4x4 matrices using AlphaTensor, aiming to determine the average runtime over five runs. The results indicate that the SIMD AlphaTensor method is significantly faster (see Fig. 4). Utilizing reinforcement learning to discover new algorithms is a groundbreaking approach. Traditionally, algorithm design has relied on manual, human-driven processes. This research highlights the potential of AI techniques in automating the discovery of new algorithms.

VII. Conclusions

Leveraging AI and reinforcement learning through the tensor game can significantly optimize matrix multiplication, driving advancements in computational efficiency and mathematical problem-solving.

References

[1] Alhussein Fawzi, Matej Balog, Aja Huang, Thomas Hubert, Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Francisco J. R. Ruiz, Julian Schrittwieser, Grzegorz Swirszcz, David Silver, Demis Hassabis. (2022). "Discovering faster matrix multiplication algorithms with reinforcement learning." Nature. Volume 609, Issue 1107, Pages 63–68. algorithms with reinforcement learning." Nature. Volume 609, Issue 1107, Pages 63–68.

[2] Bürgisser, P., Clausen, M. & Shokrollahi, A. Algebraic Complexity Theory Vol. 315 (Springer Science & Business Media, 2013).

Fig. 3. **Detailed view of AlphaTensor's architecture, included torso, policy and value head**.

Fig. 4. **Experiment Result**.

[3] Hubert, T. et al. Learning and planning in complex action spaces. In International Conference on Machine Learning 4476–4486 (PMLR, 2021).
[4] Silver, D. et al. Mastering the game of Go with deep neural networks and tree search. Nature 529, 484–489 (2016).
[5] Vaswani, A. Attention is all you need. In International Conference on Neural Information Processing Systems Vol 30, 5998–6008 (Curran Associates, 2017).
[6] Dabney, W., Rowland, M., Bellemare, M. & Munos, R. Distributional reinforcement learning with quantile regression. In AAAI Conference on Artificial Intelligence Vol. 32, 2892–2901 (AAAI Press, 2018).