4 Credit Project Report

Brendan Biernacki
CS 443

May 4, 2024

Some portions of this are copied and pasted from my proposal.

1 Background and Motivation

The paper is available here: `https://www.nature.com/articles/s41586-022-05172-4`.

The purpose of this paper is to find algorithms to multiply arbitrary sized matrices quickly. The standard algorithm takes $O(n^3)$ multiplications, but it is possible to do better. A common technique is instead to find a formula multiply fixed size matrices while minimizing the total number of multiplications, and apply this formula recursively to arbitrary sized matrices. The first improvement, the Strassen algorithm, uses $O(n^{\log_2(7)})$ multiplications, by finding a formula for 2×2 matrix multiplication requiring 7 multiplications.

The authors note the problem of minimizing multiplications is equivalent to minimizing the rank of a specific 3-dimensional tensor, T_N, encoding matrix multiplication. That is, if one can find vectors $\mathbf{u}^{(i)}, \mathbf{v}^{(i)}, \mathbf{w}^{(i)}$ that decompose T_N,

$$T_N = \sum_{r=1}^{R} \mathbf{u}^{(r)} \otimes \mathbf{v}^{(r)} \otimes \mathbf{w}^{(r)},$$

then this generates a matrix multiplication algorithm in $O(n^{\log_N(R)})$ time.

2 Problem Statement

We use an MDP $(\mathcal{S}, \mathcal{A}, P, R, \gamma)$ where:

- $\mathcal{S} = \{\text{all } N^2 \times N^2 \times N^2 \text{ tensors}\}$. We restrict the set to those tensors with entries chosen from $\{-1, 0, 1\}$. The authors used $\{-2, -1, 0, 1, 2\}$.
- $\mathcal{A} = \{\mathbf{u} \otimes \mathbf{v} \otimes \mathbf{w} \mid \mathbf{u}, \mathbf{v}, \mathbf{w} \in \{-1, 0, 1\}^{N^2}\}$ (all rank 1 tensors). The authors also used $\{-2, -1, 0, 1, 2\}$ at least in generating the synthetic data.
- Transitions are deterministic. In state T taking action $\mathbf{u} \otimes \mathbf{v} \otimes \mathbf{w}$, the new state is $T' = T - \mathbf{u} \otimes \mathbf{v} \otimes \mathbf{w}$. If $T' \notin \mathcal{S}$, there are multiple options - end the trajectory, return back to T with a nonzero penalty, return to $\mathbf{0}$ with a large penalty equal to the naive cost of decomposing a matrix (number of nonzero entries). (In my case, I just avoided these actions.) If $T = \mathbf{0}$, then we can end the trajectory or always return back to $\mathbf{0}$ with reward 0. The reward of every other (normal) state-action pair is -1.
- $\gamma = 1$. We can set a horizon of $H = N^3 - 1$ as we are only concerned with nontrivial decompositions - those that use fewer than N^3 factors.

The start state is the tensor T_N encoding the formula for matrix multiplication, possibly rotated by some change of basis matrix. Of course the goal is to find an example of as few actions as possible to reach $\mathbf{0}$.

3 Algorithms

3.1 Synthetic Demonstrations

Run many trials of the following. For each trial, we uniformly select $r \in \{1, \ldots, N^3 - 1\}$ as the rank of the tensor we would like to construct. (Though the rank may actually be smaller than r.) Construct r triples $(\mathbf{u}^{(t)}, \mathbf{v}^{(t)}, \mathbf{w}^{(t)})$ where each entry in each vector is

$$\begin{cases} -1 & \text{prob } p \\ 0 & \text{prob } 1-2p \\ 1 & \text{prob } p \end{cases}$$

and yield $\sum_{t=1}^{r} \mathbf{u}^{(t)} \otimes \mathbf{v}^{(t)} \otimes \mathbf{w}^{(t)}$ along with $(\mathbf{u}^{(1)} \otimes \mathbf{v}^{(1)} \otimes \mathbf{w}^{(1)}, \ldots, \mathbf{u}^{(r)} \otimes \mathbf{v}^{(r)} \otimes \mathbf{w}^{(r)})$.

This allows us to quickly gather (often tight) upper bounds on the ranks of many tensors. I used this as training data for my neural network.

3.2 Change of Basis

The purpose of this is to generate many different starting states. If a trajectory reaches $\mathbf{0}$ in R steps from any of these starting states, then the formulas generated for these tensors can then be adapted for the original T_N without changing its complexity, R. They state, "This crucial step injects diversity into the games played by the agent." If the agent gets stuck with one starting state, we can just try a different one.

The actual details of generating the matrices aren't too relevant but they are included in my code. The important feature is that they are unimodular meaning that their inverses are each integer matrices. Thus, the alternative start states will (likely) be in our state space.

The authors ran trajectories from many starting states in parallel, whereas I was unable to take advantage of using multiple threads, possibly due to Python or Google Colab limitations. Thus, I did not benefit much from using this.

3.3 Neural Network

While the authors provided a clear description of their neural network, it was too complicated for me to implement given my knowledge. I didn't realize how challenging this component would be to replicate.

I had also considered the libraries for stable implementations but I was unable to successfully model the environment and I was unsure if they would even help since the states and actions are discrete but not feasibly enumerable.

Since this is a critical component of the learning algorithm, I decided to try to teach a simpler neural network to learn as close of an approximation to $V^\star$ as possible. If at a state s, an action leads to a state s' whose estimate is better than expected: $V^\star(s') \gtrsim V^\star(s) + 1$, this action would possibly help construct a decomposition with fewer factors. The expected increase for a typical correct action would be $+1$; if an action leads to an increase of $+2$, then it is likely the neural network estimated $V^\star(s)$ incorrectly and the current sequence of actions is part of an improved trajectory.

One note of positivity about this problem is that we don't need to optimize the expected return, we simply need to find one successful trajectory. Another point worth mentioning is that from the synthetic demonstrations, it is very easy to get more training data.

4 Experiment

4.1 Setup

All of the experiment was done in a Google Colab notebook here: `https://colab.research.google.com/drive/1r0AfWe5RnR-7hoQkDXyaw2ALJEpnSf4k?usp=sharing`. I am also uploading the relevant files to a respository here: `https://github.com/brendanb321/MockAlphaTensor.git`. I mainly used the TPU and TPU v2 processors (and the standard CPU when I reached the usage limits).

The experiment was mostly intended to independently find an algorithm like Strassen's - a decomposition of T_2 into 7 rank-1 tensors.

4.2 Results

As of the time of writing this, I have not independently found any decompositions for T_2 into 7 rank-1 tensors. In the (unlikely) event I happen to find one, I will upload it to the repository.

I believe the likely reasons are that I did not use their Monte-Carlo Tree Search algorithm and my neural network isn't sufficient. The only learning I implemented was estimating $V^\star$. I was able to teach a much simplified version of their neural network to an average loss of about 0.70, however this only considered the current state and the training data wasn't guaranteed to be correct. Thus, it would be difficult to detect whether taking an action actually improved $V^\star$ by the necessary amount. Since I didn't implement their version of policy evaluation (MCTS), my choices for actions were probably near random.

I tested my code using the decomposition for Strassen's algorithm to ensure I didn't have any bugs.

4.3 Analysis

Generally speaking, the assumptions I made were too naive to reduce the search space enough.

More concretely, I found that there are $|\mathcal{A}| = 128001$ actions in the 2×2 case. Guessing 7 actions correctly has a $\frac{1}{|\mathcal{A}|^7} = 1.77 \times 10^{-36}$ chance, assuming a brute force approach.

With my implementation, my neural network doesn't help me algorithmically reduce the action space significantly at each state, but suppose it correctly reduces the action space by a factor of 16 to be generous. (This the number of actions I test at a given state and greedily select the best one according to my neural network.) We are able to permute the actions in 7! ways, and we can immediately tell if the s_6 is a rank-1 tensor. (Precomputing all rank-2 tensors used too much RAM.) Assuming Strassen's algorithm is the only possible decomposition with 7 factors (up to change of basis, scaling, permutations, etc.), this puts the probability of any trial succeeding as very roughly

$$\left(\frac{16}{|\mathcal{A}|}\right)^6 \cdot 7! = 1.92 \times 10^{-20},$$

a good improvement, but not near close enough. For reference, I was able to run almost exactly 100000 trials each hour.

4.4 Follow-up Experiment

Taking inspiration from the modulo 2 idea, I decided to try to search for a 2×2 matrix multiplication formula where the entries were taken modulo 2. I would then determine the correct signs on the 1's to extend this to a formula over the integers once I had a correct formula modulo 2.

This reduced the search space significantly, to $|\mathcal{A}| = 3376$. I was able to fit all rank-2 tensors in RAM using some bit tricks. I also removed the neural network since it was too slow and wasn't improving the policy by more than a small factor. Thus, this was a pure brute-force attempt with no RL. I also removed the change of basis start state since I hadn't been using it to my advantage.

With these optimizations, I estimate the probability of success as

$$\frac{1}{|\mathcal{A}|^5} \cdot \frac{7!}{2} = 5.75 \times 10^{-15},$$

where I was able to run about 135 million trials per hour. This is a little conservative, since I only chose actions that could not combine with a previously taken action to form a rank-1 tensor, but it's still not great.

5 Conclusions

It shouldn't be too surprising that all of the AlphaXXX agents by DeepMind are receiving so much attention. They invested a lot of time and money into using their computing resources and developing deep neural networks for predicting optimal actions. I'm not surprised I was unable to reproduce their results, not to be humble. There's a reason this approach to this problem was only developed within the past 2 years. Finding algorithms that do not require exhausting the entire search space remains difficult.

RL is hard. I have a lot of respect for you, Prof. Jiang, to do research in this field.

6 References

- Original published article. https://www.nature.com/articles/s41586-022-05172-4
- Supplementary information. Helped with change of basis and synthetic demonstration details. https://static-content.springer.com/esm/art%3A10.1038%2Fs41586-022-05172-4/MediaObjects/41586_2022_5172_MOESM1_ESM.pdf
- Publicly provided code which was not helpful for recreating the experiment. It's more helpful if someone wanted to verify their results. https://github.com/deepmind/alphatensor
- Talk by Matej Balog, one of the authors. https://www.youtube.com/watch?v=QJT3LszR2pA
- Simplified explanation by Emma Chen, a PhD student at Harvard. https://www.youtube.com/watch?v=gpYnDls4PdQ
- Video by Quanta Magazine, introducing me to the paper. https://www.youtube.com/watch?v=fDAPJ7rvcUw
- Subsequent paper improving some results over the original article. https://arxiv.org/pdf/2210.04045.pdf
- PyTorch documentation.
- Gym documentation. Not too helpful.
- Stable Baselines documentation. Not too helpful.
- Video on the overview of this matrix multiplication problem (not just in an RL setting). https://www.youtube.com/watch?v=JAb1Kj3vYCQ