Crafting efficient reward function for Causal Discovery RL

Baraa Al Jorf, Dmytro Zhylko

December 2023

1 Introduction

In the age of terabyte-scale multi-dimensional datasets and extensive employment of large blackbox models to harness data scale for predicting natural phenomena, the deep learning community is increasingly drawn to causal inference as an intuitive and efficient means of incorporating prior knowledge. It also serves as a way to explicitly model inductive bias within black box estimators, enhancing trustworthiness and explainability of models. However, a substantial challenge in the realm of causal inference lies in establishing causal graphs, typically represented as Directed Acyclic Graphs (DAGs), connecting causes with effects.

Traditional manual construction of causal DAGs with domain experts becomes unfeasible in modern deep learning settings. Unlike classical approaches, contemporary DAGs may involve thousands or even hundreds of thousands of variables, and the requisite domain expertise may be lacking. For instance, complex systems like protein signaling networks pose challenges, where ongoing research continuously unveils underlying interactions, making it impossible for a "domain expert" oracle to generate the causal graph. Consequently, the field of Causal Discovery has witnessed significant growth, particularly in the realm of Causal Discovery with Reinforcement Learning (RL). RL algorithms leverage, potentially stochastic, reward functions to guide learning agents toward optimal action policies, with the agent's actions effectively constituting the construction of the causal graph.

Within the scope of this paper, our focus is on devising a **practical** approximation for the existing reward function used in training causal discovery models. We discern weaknesses and limitations in popular methods, proposing interesting research questions that can be area of further research. The culmination of our efforts is presented through empirical findings and discussions in the subsequent sections of this report.

2 Related work

In the historical landscape of causal discovery methods, prevalent approaches relied on exhaustive combinatorial search. This involved conditional independence tests that pruned the full graph or score-based functions that greedily added edges to improve the graph's score. However, the worst-case exponential runtime of these methods rendered them impractical for addressing intriguing large-scale problems. This limitation garnered attention from the continuous optimization community, culminating in the development of the first continuous optimization algorithm for causal discovery known as NOTEARS [3]. Subsequently, a plethora of methods emerged, most altering the family of functions considered in the structural equations model (SEM) while keeping the problem formulation largely unchanged.

A crucial aspect of the design is framing the problem of causal discovery as a constrained optimization problem, with the constraint function $h(A)$ defining Directed Acyclic Graphs at the zero level set, meaning $h(A) = 0 \iff G(A) \in DAG$. NOTEARS introduced criteria for the constraint function, requiring it to be a quantification of the "DAG-ness" of the weighted graph spanned by A, smooth and possessing easy-to-compute derivatives. The initial formulation of $h(A) = tr(I - A)^{-1} - d$ proved numerically unstable, leading to the formulation $h(A) = e^A$, which provided greater numerical stability, while defined in similar fashion as trace of power-series of the matrix, functionally counting discarded sum of weighted self-loops.

However, our focus shifts to another noteworthy work [4], which employs a Reinforcement Learning (RL) algorithm to perform a "relax and round" combinatorial search akin to classical approaches with a score function. The reward function in this method is defined as:

$$\begin{aligned} \text{reward} &:= -[S(G) + \lambda_1 I(G \notin DAGs) + \lambda_2 h(A)], \\ h(A) &:= \text{trace } e^A - d. \end{aligned}$$

Here, S is typically chosen as the Bayesian Information Criterion (BIC) score, evaluating the fitness of a causal model to observed data. The BIC score is formulated as:

$$SBIC(G) = md \log \left(\frac{1}{md} \sum_{i=1}^{d} RSS_i \right) + \#(\text{edges}) \log m.$$

Notably, the reward function in this context doesn't necessarily have to be differentiable or deterministic, as RL algorithms optimize for the expected reward. This flexibility allows room for improvement upon the most computationally expensive operation in the reward function, which is e^A. With a computational complexity of $O(N^3)$ and a relatively large constant, this operation significantly impacts real-world execution time, given that the reward function is called every iteration of optimization and RL's notorious batch inefficiency over thousands of epochs.

2.1 Problem analysis

We first need to understand within what bounds we can optimize our matrix exponential. We previously stated that our function doesn't have to be neither differentiable, nor deterministic, so we can employ randomized strategies to compute said value. We also should analyze family of matrices over which we want our method to be defined, for that we look into decoder part of RL agent architecture, because it's the only relevant part. Agent outputs the adjacency matrix with following entries:

$$A_{ij} = \sigma(u^T tanh(W_1 enc_i + W_2 enc_j)),$$

where $W_1, W2_2 \in \mathbf{R}^{d_h \times d_e}, u \in R^{d_h \times 1}$ are trainable parameters and σ - logistic function. Which for the inference porpoises is then converted to binary matrix with Bernoulli sampling with probability A_{ij} and rounded to ensure a DAG. Also considering the fact, that default initialization strategy for torch framework is uniform Xavier, we know that initial model guesses will be similar to random Erdős–Rényi graph with $p = 0.5$ and in later stages we want agent to produce DAG. From perspective of adjacency matrix we are converging from dense matrix with random positive entries $\in [0, 1]$, towards relatively sparse asymmetric matrix with positive entries. So we need method that would be robust enough to efficiently handle both dense and sparse graphs. Unfortunately, we cannot theoretically analyze this problem further, since it would require to make assumptions on training dynamics, instead we opt to empirically evaluate performance of proposed methods.

3 Methods

To design a more efficient reward functions, we utilize a few trace and exponential approximations and calculation methodologies presented in [2] and hand crafted methods we developed ourselves. Despite good runtime guarantees of several methods like Lanczos or other Krylov subspace methods for both eigenvalue or matrix exponent estimation, we cannot apply most of them, since they are only defined for symmetric problems, where we are specifically targeting asymmetric matrices. Each presented method has its own computational requirements and assumptions, which we present in next sections.

3.1 Naive Estimation(Taylor's Series)

We first refer to basic definition of matrix exponent derived from the power series, from which we know that $e^A = \sum_k \frac{1}{k!} A^k$. This however doesn't improve the computational complexity, since we still have to compute matrix-matrix multiplications. One way to overcome this issue, that we exploited in this work

is to apply Hutchinson Trace Estimation [1]:

$$trace(e^A) = trace\left(\sum_k \frac{1}{k!}A^k\right) \quad = \sum_k trace\left(\frac{1}{k!}A^k\right) \approx \sum_k \frac{1}{k!}\frac{1}{|V|}\sum_{v\in V} v^T A^k v$$

We know that this method is not well regarded because of convergence issues, that arise when matrix has large negative eigenvalues or big eigenvalue span, however referring to Problem Analysis, we know that this is not the case for our problem. Moreover we can limit number of iterations significantly since very long cycles contribute little to overall sum due to factorial term and small scale of matrix entries, so we can limit number of iterations to $k = 10$, when $k! > 10^6$. In addition this method is very easy to optimize since implementation only involves "vectorizable" operations and can be expressed as:

$$h(A) = \sum_k \frac{1}{k!|V|}\sum VA^k \odot V,$$

where $B = VA^{k-1}$ can be saved from previous iteration, so only $\sum BA \odot V$ should be computed every iteration. With final runtime of the method being $O(n^2 k|V|)$.

3.2 Deterministic heuristic algorithmic approach: Shortest loops

In this approach we analyze our problem from different perspective, specifically we start with graph interpretation of adjacency matrix A raised to the power k. Trace of matrix A^k results in counting number of all self loops of length exactly k in graph that follows A adjacency, note that for weighted graph loops are weighted. We then can craft graph heuristic that uses graph representation to compute cyclicness score of the given graph. Our heuristic should satisfy following criterion $h(A) = 0 \iff G(A) \in DAG$. For this we devised simple heuristic: for every node run BFS to find first shortest self loop (if exists) and save it's weight. We then use the fact that shortest loop will repeat on lengths of the walk that are proportional to length of the loop with proportional weight, this is why we compute the sum of accordingly scaled weights up to length of the walk k which is a hyperparameter of the method. Assumption here is that shortest path with all of it's repetitions will contain big enough proportion of the overall sum. We see that this heuristic satisfies $h(A) = 0 \iff G(A) \in DAG$, since if graph is acyclic we won't dint any self-loops. And we also see that heuristic in some degree detours presence of self-loops with large weights and grows with the number of loops just like original reward function. One could also utilize Dijkstra's algorithm to find cycles of highest weight with marginal change in computational complexity. Presented heuristic has one desired property of being almost embarrassing parallel with only weight and length storing being synchronous. However, it's worth noticing that operations used for BFS are less heavily optimized and at worst case method also achieves cubic runtime since

it's calling N BFS sub-routines with $O(N^2)$ worst case complexity, but in all fairness this cubic runtime comes with very small constant factor.

3.3 Schur Decomposition

The Schur method for calculating the matrix exponential e^A is particularly efficient for dense matrices. It begins with the Schur decomposition, expressing matrix A as QTQ^T, where Q is orthogonal matrix and T is upper triangular with eigenvalues on the main diagonal. We then can utilize equation $trace\left(e^A\right) = \sum_i e^{\lambda_i}$, so $trace\left(e^A\right) = \sum e^{diag(T)}$. This method is exact solution to the matrix exponent problem, however it's very well suited for practical settings, since it's defined for arbitrary matrix and has very efficient subroutines in common computational libraries.

4 Results and Conclusions

We compare performance of algorithms both in terms of real execution time and convergence of original RL algorithm on simulated data detailed in [4].

4.1 Runtime

We attempt fair comparison between different methods, however we do recognize that evaluation results might be different depending on processor family and environment setup. To mitigate unfair advantage of compiled scipy implementations of schur and expm (original), we utilize just-in-time compilation provided by numba package. In order to make BFS even more appealing we, in addition to JIT-ing, utilize OMP (Open MP) simple loop parallelization on very top level (for each node), to match advantage of other methods in utilization of multi-core system. Note: JIT while bridging the gap between scipy and python might in turn skew results towards naive method, by compiling it with vector instructions available on most contemporary systems, that might not have been baked to generic scipy subroutines. Nevertheless we proceed with our evaluation with this note serving as a warning of potential bias. We present time obtained from evaluations on random DAG with edge probability $p = 0.2$ (as in simulated data) all measured by ipython's "magic" function "%timeit" in Table 1, we onlu evaluate for big graphs since otherwise measurments would we unreliable. We only evaluate for DAG since it's most difficult case for BFS heuristic, since it has no early stopping with no self loops. We evaluate on 2018 Mac book pro, with Intel Core i7-8750H. However we only demonstrate the scale of numbers in this table since real execution time strictly followed this ordering: Origina > Schur > Naive > BFS, with BFS being the fastest one amongst all the methods or average.

Size	Original	Naive	Schur	BFS
200	1.44 ms ± 100 μs	1.72 ms ± 103 μs	434 μs ± 9.74 μs	663 μs ± 8.86 μs
500	20.6 ms ± 930 μs	5.31 ms ± 81.4 μs	3.43 ms ± 14.2 μs	7.96 ms ± 406 μs
1000	160 ms ± 7.75 ms	16.1 ms ± 998 μs	26.8 ms ± 387 μs	50.9 ms ± 1.68 ms

Table 1: Timing for each method execution on random DAG.

4.2 Evaluation on simulated data

In this section we present results of actual run for presented methods against original implementation measured with 5 different seeds for increasing problem size in Tables 2, 3, 4, 5. For unknown reasons original framework was not able to run further than 20 edge graph resulting in OOM Kill, which might have been an issue in latest PyTorch installation potentially causing memory leak, since issue disappeared after installing version 1.13.1, however as results taking hours to compute, we did not have time to obtain evaluation for bigger matrices.

Data was generated according to original paper [4] and utilized modified Erdős–Rényi scheme to generate DAG with $p = 0.2$, which was then used to produce data according to implied distribution. We report false discovery rate (fdr), true positive rate (tpr) and structural hamming distance (shd) as main evaluation metrics. As we can see our heuristics hold their own in this task, however we can clearly observe naive method unproportional deterioration with size increase, which might indicate that we needed to increase number of iterations for power-series to yield better results. One surprising finding was that BFS is actually able to sometimes outperform other methods. However small scale of the experiment doesn't allow to draw definitive conclusions about superiority of any of the methods estimation methods.

Original Method			
Size	fdr	tpr	shd
10	0. ± 0.	1. ± 0.	0. ± 0.
15	0.03 ± 0.05	0.99 ± 0.03	0.5 ± 0.87
20	0.14 ± 0.17	0.88 ± 0.13	7.75 ± 8.35

Table 2: Original Method Evaluation Results on Simulated Data

Naive Results			
Size	fdr	tpr	shd
10	0.22 ± 0.16	0.93 ± 0.12	2.25 ± 1.48
15	0.24 ± 0.09	0.85 ± 0.12	5.75 ± 2.59
20	0.34 ± 0.09	0.73 ± 0.12	20.67 ± 6.34

Table 3: Naive Method Evaluation Results on Simulated Data

Schur			
Size	fdr	tpr	shd
10	0. ± 0.0	1. ± 0.	0. ± 0.
15	0.08 ± 0.05	0.99 ± 0.03	0.5 ± 0.87
20	0.17 ± 0.09	0.88 ± 0.14	8.5 ± 5.03

Table 4: Schur Method Evaluation Results on Simulated Data

BFS			
Size	fdr	tpr	shd
10	0. ± 0.	1. ± 0.	0. ± 0.
15	0. ± 0.	1. ± 0.	0. ± 0.
20	0.24 ± 0.05	0.88 ± 0.07	11.25 ± 2.39

Table 5: BFS Method Evaluation Results on Simulated Data

References

[1] Michael F Hutchinson. A stochastic estimator of the trace of the influence matrix for laplacian smoothing splines. *Communications in Statistics-Simulation and Computation*, 18(3):1059–1076, 1989.

[2] Cleve Moler and Charles van Loan. Nineteen dubious ways to compute the exponential of a matrix, twenty-five years later. *SIAM Review*, 45(1):3–49, 2003.

[3] Xun Zheng, Bryon Aragam, Pradeep K Ravikumar, and Eric P Xing. Dags with no tears: Continuous optimization for structure learning. *Advances in neural information processing systems*, 31, 2018.

[4] Shengyu Zhu, Ignavier Ng, and Zhitang Chen. Causal discovery with reinforcement learning. *arXiv preprint arXiv:1906.04477*, 2019.