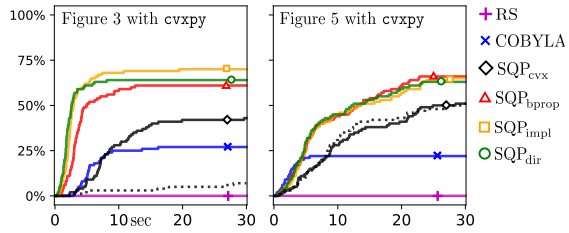
Dear reviewers, first we want to say that your feedback was very helpful. We see a few genuine concerns and some misunderstandings, so we appreciate the opportunity to address them. Specific issues raised by the reviewers (R1, R2, R3, R4) have been organized into paragraphs below. Thank you! — The Authors.

***The gradient-free baseline should be (e.g.) BARON, not COBYLA* (R2).** BARON is not applicable: a gradient-free baseline must support either bilevel natively or must allow black-box objective and constraints (i.e., callbacks). This also rules out CPLEX and Gurobi. IPOPT, BONMIN and Couenne are all gradient-based. YALMIP (MATLAB) supports bilevel via reformulation as a single-level model, but imposes restrictions on how parameters appear in the inner problem. Pyomo's `bilevel` support is restrictive and deprecated. We would like to clarify that our aim was to compare *local* NLP methods like COBYLA and SQP, not global methods like BARON, ARGONAUT [Boukouvala et al., 2016], or DOMINO [Beykal et al., 2020]. Of the many gradient-free solvers provided by the well-known NLopt package, the documentation explains that only COBYLA supports arbitrary non-linear equality and inequality constraints. (Likewise, the only such gradient-based solver in NLopt happens to be SLSQP.) Finally, note that COBYLA was used as a state-of-the-art local NLP baseline in the ARGONAUT and and DOMINO papers.

***Compare to OptNet/CvxpyLayers* (R2, R3).** We agree that such a comparison is important for the gradient computation component of our work. In fact, method SQP_{impl} (implicit differentiation) in our experiments was intended to be this comparison: it is our efficient implementation of OptNet specialized to LPs, and performed well. However, as `cvxpylayers` [Agrawal et al. 2019] can implement OptNet efficiently, we agree that a comparison to that implementation is informative. We used `cvxpylayers` to implement method SQP_{cvx}, shown at right. As in our previous experiments, we use the same vectorized LP generation, solve in batch mode, enable efficient re-solves, and use double precision. We confirmed that timing differences were due to the speed of the internal `scs` solve (fast conic solver written in C) and not any overhead. The only difference is that the default `max_iters` for `cvxpylayers` can terminate long before reaching the requested precision (10^{-8}, the SciPy default), but we were careful to ensure that this did not limit its maximum achievable success rate. We found `scs` can be very slow when held to the same precision standard as the rest of our experiments (dotted line). We in no way "relied upon" [Agrawal et al. 2019] in developing our work.



***Compare to [Tan et al. 2019]* (R1).** A comparison with their work was done but not highlighted due to two reasons: their method is not applicable to objective error loss when learning cost with constraints, as it returns arbitrarily bad solutions like $\mathbf{w}_1$ in Figure 1 of the paper; their forward solver implementation is much slower and occasionally fails.

***Compare to specialized inverse methods* (R2, R3).** To the best of our knowledge, there are no other inverse optimization (IO) methods in the literature that can be directly applied to the problem instances we investigated. Most IO methods make strong assumptions or address one aspect of the problem. Our approach is qualitatively more flexible.

***Scalability concerns* (R2).** We are actually very excited about the scalability opportunities of our approach! First, consider that the forward solver can be any specialized algorithm that returns primal and dual variables for the problem at hand, not just a general-purpose LP solver; this means the forward solve takes a much smaller fraction of compute. Second, with fast forward solvers the speed advantage of our closed form gradient becomes important, because solving a large system (OptNet/CvxpyLayers-style) scales much worse than simply computing an outer product (our approach); even on these modest instances, our closed form gradients are >100x faster than `CvxpyLayer.backward`. Other well-known IO approaches, such as single-level models [Keshavarz et al., 2011], cannot exploit fast solvers for the forward problem (i.e., the inner problem of our formulation) and must rely on general-purpose machinery like CPLEX.

***Isn't Theorem 1 a special case of Appendix B in [Agrawal et al., 2019]?* (R3).** No. Although LPs are a special case of convex programs, our theorem applies only to objective error loss (not considered by [Amos & Kolter, 2017] nor [Agrawal et al., 2019]) and cannot be derived by merely specializing the steps of their existing proofs.

***Prove SQP convergence to second-order stationary points* (R2).** We agree that such a proof would be informative, but characterizing the theoretical properties of a particular gradient-based NLP method (such as SQP) in this setting was not our goal. Different parametrizations may or may not satisfy the technical conditions required (e.g., to guarantee that $\mathbf{B}$ matrix remains a good Hessian estimate), but such analysis was not the focus of our work.

***The paper just applies SQP to an NLP, which is obvious* (R2).** The IO formulation itself is our contribution and, despite seeming "natural" (we agree!), has not been proposed before in either the `cvxopt`/OptNet body of work nor in the IO-specific literature. So, the paper is not just about "applying SQP to an NLP." Our framework provides the most general approach for inverse LPs to date. Further, due to the importance of LPs and the existence of many fast algorithms for specialized LPs, we believe that our fast gradient formula has long-term value.