Explanation of the Paper Based on Code and Replication Experience

Abstract

The paper, *Dynamic Causal Bayesian Optimization (DCBO)*, presents a novel approach for conducting a sequence of optimal interventions in causal dynamical systems. This approach combines causal inference, Gaussian Process (GP) emulation, and Bayesian Optimization (BO) to optimize target variables over time while accounting for temporal causal dependencies.

Insights from the Code

The codebase provides a complete implementation of the methodologies described in the paper, with several aspects extending or elaborating on the theoretical framework. Key insights include:

1. Dynamic Causal GP Model

- The paper briefly describes using a "dynamic causal GP model" to quantify uncertainty and predict outcomes under interventions.
- The code implements this through the `CausalRBF` kernel in `causal_kernels.py`, which adapts standard RBF kernels to account for causal dependencies. This detail is absent in the paper.

2. Data Transfer Across Time Steps

- The paper mentions transferring interventional knowledge across time steps but provides little implementation detail.
- The code operationalizes this with functions like `update_sufficient_statistics` in `root.py` and `update_bo_model` in `dcbo_base.py`, which iteratively update GPs using observational and interventional data.

3. Acquisition Functions

- While the paper provides the theoretical background, the central acquisition function, *Causal Expected Improvement (CEI)*, is concretely implemented in `evaluate_acquisition_f` in `intervention_computations.py`.

- This function handles dynamic updates, constraints, and numerical optimization, aspects not elaborated in the paper.

4. SEM Estimation

- Structural Equation Models (SEMs) are the causal backbone of DCBO. The estimation routines in `sem_estimate.py` use GPs to model causal relationships dynamically.
- These details are crucial for replicating experiments but are not discussed in the paper.

5. Extensions and Practical Implementations

- Tutorials in `notebooks/`, such as `nonstat_scm.ipynb`, provide practical insights into running experiments on complex SCMs, which are not covered by the examples in the paper.
- The integration of multiple BO variants (ABO, BO, CBO) in the code showcases comparative performance across settings, giving insights not fully explored in the paper.

Replication Challenges and Insights

1. Translation to TensorFlow Probability (TFP)

- The paper does not specify any GP library, but the code heavily uses GPy. Rewriting the code in TFP required nontrivial adaptations, such as implementing custom kernels (`CausalRBF`) and GP fitting functions (`fit_gp`).
- Numerical differences arose during replication due to discrepancies in kernel definitions and optimization algorithms between GPy and TFP.

2. Paper Omissions

- Several auxiliary functions and configurations, such as SEM fitting, DAG manipulation, and integration tests, were crucial for replication but not discussed in the paper.
- For instance, the parameters of the `CausalRBF` kernel and acquisition function optimizations required significant adjustments in the TFP implementation.

3. Testing Framework

- Extensive testing, including unit tests, integration tests, and regression tests, ensured correctness during replication.
- These tests, not mentioned in the paper, provided insights into corner cases and numerical stability.

Conclusion

Despite successfully translating the GPy-based dependencies to TensorFlow and TensorFlow Probability, and passing all unit, integration, and regression tests, challenges were encountered in replicating the results from the original paper. Specifically, while running the `nonstat_scm.ipynb` notebook, errors consistently occurred during the execution of key functions related to updating sufficient statistics and handling dynamic SEM updates. These issues indicate potential gaps in the integration or management of dependencies within the codebase.

It would be highly beneficial if the authors provided a clear schematic representation of the dependency flow and execution pipeline of the code. Such documentation would greatly enhance the understanding of the interactions between various components and facilitate smoother adaptation and debugging, especially for researchers aiming to implement the framework using alternative libraries like TensorFlow Probability.