DiffOpt: Parallel optimization of Jax models

Alan N. Pearl [1], **Gillian D. Beltz-Mohrmann** [1], **and Andrew P. Hearin** [1]

1 HEP Division, Argonne National Laboratory, 9700 South Cass Avenue, Lemont, IL 60439, USA

DOI: 10.21105/joss.07522

Software

- Review
- Repository
- Archive

Editor: Mehmet Hakan Satman

Reviewers:

- @landreman
- @ewu63

Submitted: 15 October 2024
Published: 17 December 2024

Summary

`diffopt` is a Python package which facilitates in the optimization of data-parallelized, differentiable models using the Jax (Bradbury et al., 2018) framework. It is composed of three subpackages, `multigrad`, `kdescent`, and `multiswarm`. Leveraging MPI (Message Passing Interface), `multigrad` efficiently sums and propagates gradients of custom-defined summary statistics across processors and computing nodes. `kdescent` utilizes mini-batched kernel density estimates to perform stochastic gradient descent to fit a full model distribution to an N-dimensional training dataset. A massively parallelizable implementation of particle swarm optimization (PSO) is provided by `multiswarm`, enabling global optimization of even high-dimensional, non-convex loss surfaces. Our simple yet flexible design makes these methods applicable to a wide variety of problems requiring solutions scalable to large amounts of data through both gradient- and non-gradient-based optimization techniques. Visit our documentation page to learn the usage.

Statement of Need

In and beyond the field of cosmology, parameterized models can describe complex systems, provided that the parameters have been tuned adequately to fit the model to observational data. Fitting capabilities can be increased dramatically by gradient-based techniques, particularly in high-dimensional parameter spaces. Existing gradient descent tools in Jax do not inherently support data-parallelism with MPI, creating a speed and memory bottleneck for such computations.

`multigrad` addresses this need by providing an easy-to-use interface for implementing data-parallelized models. It handles the MPI reductions as well as the mathematical complexities involved in propagating chain rules required to compute the gradient of the loss, which is a function of parallelized summary statistics, which are in turn functions of the model parameters. At the same time, it is very flexible in that it allows users to define their own functions to compute their summary statistics and loss. As a result, this package can enable scalability through parallelization to the optimization routine of nearly any big-data model. `kdescent` and `multiswarm` each provide powerful fitting tools which are fully compatible with the parallelization framework laid out by `multigrad`.

Past efforts have already been made towards parallelization of Jax (`mpi4jax`, Häfner & Vicentini, 2021), parallel gradient descent (e.g., Gray et al., 2019), and parallel PSO (Blank & Deb, 2020; Li & Wada, 2005). Our approach combines many of these features and more into one easy-to-use, documented Python module with all the tools to optimize arbitrarily complex models that the user has implemented in the Jax framework. Additionally, while the fundamental MPI reductions available through `mpi4jax` are generally sufficient, our `multigrad` procedure provides significant convenience for problems in which complex summary statistics are computed in parallel before being applied to a differentiable loss function.

Method

`multigrad`

`multigrad` allows the user to implement a loss term, which is a function of summary statistics, which are functions of parameters, $L(\vec{y}(\vec{x}))$ where the summary statistics are summed over multiple MPI-linked processes: $\vec{y} = \sum_i \vec{y}_{(i)}$ where i is the index of each process. In this section, we will derive the gradient of the loss $\vec{\nabla} L$ with respect to the parameters and as a sum of terms that each process can compute independently.

We will begin from the definition of the multivariate chain rule,

$$\frac{\partial L}{\partial x_j} = \sum_k \frac{\partial L}{\partial y_k} \frac{\partial y_k}{\partial x_j}$$

where $\partial y_k = \sum_i \partial y_{k(i)}$. By pulling out the MPI summation over i,

$$\frac{\partial L}{\partial x_j} = \sum_i \sum_k \frac{\partial L}{\partial y_k} \frac{\partial y_{k(i)}}{\partial x_j}$$

and by rewriting this as vector-matrix multiplication,

$$\vec{\nabla}_x L = \sum_i (\vec{\nabla}_y L)^T J_{(i)}$$

we can clearly identify that each process has to perform a vector-Jacobian product (VJP), where $J_{(i)}$ is the Jacobian matrix such that $J_{kj(i)} = \frac{\partial y_{k(i)}}{\partial x_j}$. Fortunately, this is a computation that Jax can perform very efficiently, without the need to explicitly calculate the full Jacobian matrix by making use of the `jax.vjp` feature, saving us orders of magnitude of time and memory requirements.

`kdescent`

Mini-batching techniques often compute the loss function with only a small subset of the training data taken into account. In `kdescent`, the density of the full training dataset is measured around a "mini-batched" sample of kernel centers, which are drawn from points in the training data. With each iteration of stochastic gradient descent, a new sample of (20 by default) kernels is selected at positions $\vec{\mu}_k$ for each kernel k.

Using the `compare_kde_counts` method, the "true" and "model" counts are each computed around each kernel using the same equation below, where x_i is the i^{th} point in the training data or model data, respectively:

$$N_k = \sum_i \mathcal{N}(\vec{x}_i \mid \vec{\mu}_k, \Sigma)$$

where $\mathcal{N}$ is the multivariate-normal distribution with mean $\vec{\mu}_k$ and covariance matrix Σ (where the covariance is calculated using Scott's rule for kernel density estimation of the training dataset; Scott (1992)). It is then up to the user to define their own loss function comparing the counts of $N_{k,\text{truth}}$ to $N_{k,\text{model}}$. Note that these are extrinsic quantities (as is necessary to be parallelizable through `multigrad`) which can be reduced to intrinsic quantities for PDF-level comparisons by simply dividing by the total number of training and model data, respectively.

The analogous `compare_fourier_counts` method can provide additional loss terms relating to differences in the empirical characteristic function (ECF; Cramer (1954)). It is evaluated at a random sample of (20 by default) Fourier-space positions, $\vec{\tilde{x}}_k$, for both the "true" and "model" Fourier counts:

$$\tilde{N}_k = \sum_i \exp(i\vec{\tilde{x}}_k \cdot \vec{x}_i).$$

`multiswarm`

Particle swarm optimization (PSO; Kennedy & Eberhart (1995)) is a highly exploratory fitting algorithm in which a set of (100 by default) particles are initialized with randomized velocities and positions with Latin-Hypercube spacing over the loss function's parameter space. Each particle has an inertial weight ($w_I = 1$ by default), a cognitive weight, ($w_C = 0.21$ by default), and a social weight, ($w_S = 0.07$ by default). The default parameters have been hand-tuned to optimize parameter exploration performed by 100 particles before converging over roughly 100 time steps in a 4D Ackley loss function demonstrated in our documentation.

Within each PSO iteration: (1) Each particle's position is updated according to its current velocity $x_{i+1} = x_i + v_i$. (2) Positions and velocities are then reflected accordingly across any axes in which they have left the boundaries, if applicable. (3) Finally, the particle's velocity is slightly pulled in the direction of its personal best $x_{\rm PB}$ and global best $x_{\rm GB}$ loss found, according to the following equation:

$$v_{i+1} = w_I v_i + w_C(x_{\rm PB} - x_{i+1}) + w_S(x_{\rm GB} - x_{i+1})$$

The `multiswarm` implementation of PSO allows users to conveniently distribute the loss function computations performed by each particle across MPI ranks. Particles are evenly distributed across all ranks by default, but users needing further control can provide a custom MPI communicator object, and/or specify the `ranks_per_particle` argument to manually control intra-particle parallelization.

Science Use Case

`diffopt` was developed to aid in parameter optimization for high-dimensional differentiable models applied to large datasets. It has enabled the scaling to cosmological volumes of a differentiable forward modeling pipeline which predicts galaxy properties based on a simulated dark matter density field (Diffmah: Hearin et al. (2021); Diffstar: Alarcon et al. (2023); DSPS: Hearin et al. (2023)). Ongoing research is currently utilizing `diffopt` to optimize the parameters of this pipeline to reproduce observed galaxy properties (e.g. Beltz-Mohrmann et al. in prep.). More broadly, `diffopt` has useful applications for any scientific research that focuses on fitting high-dimensional models to large datasets and would benefit from computing parameter gradients in parallel.

Acknowledgements

This work was supported in part by the OpenUniverse effort, which is funded by NASA under JPL Contract Task 70-711320, 'Maximizing Science Exploitation of Simulated Cosmological Survey Data Across Surveys', and by the DOE contract DE-AC02-06CH11357. We gratefully acknowledge the HPC resources operated by the Laboratory Computing Resource Center at Argonne National Laboratory.

References

Alarcon, A., Hearin, A. P., Becker, M. R., & Chaves-Montero, J. (2023). Diffstar: a fully parametric physical model for galaxy assembly history. *518*(1), 562–584. https://doi.org/10.1093/mnras/stac3118

Blank, J., & Deb, K. (2020). Pymoo: Multi-objective optimization in Python. *IEEE Access*, *8*, 89497–89509. https://doi.org/10.1109/access.2020.2990567

Bradbury, J., Frostig, R., Hawkins, P., Johnson, M. J., Leary, C., Maclaurin, D., Necula, G., Paszke, A., VanderPlas, J., Wanderman-Milne, S., & Zhang, Q. (2018). *JAX: Composable transformations of Python+NumPy programs* (Version 0.3.13). http://github.com/google/jax

Cramer, H. (1954). *Mathematical methods of statistics*. Princeton Univ. Press. https://cds.cern.ch/record/107581

Gray, J. S., Hwang, J. T., Martins, J. R. R. A., Moore, K. T., & Naylor, B. A. (2019). OpenMDAO: An open-source framework for multidisciplinary design, analysis, and optimization. *Structural and Multidisciplinary Optimization*, *59*(4), 1075–1104. https://doi.org/10.1007/s00158-019-02211-z

Häfner, D., & Vicentini, F. (2021). mpi4jax: Zero-copy MPI communication of JAX arrays. *Journal of Open Source Software*, *6*(65), 3419. https://doi.org/10.21105/joss.03419

Hearin, A. P., Chaves-Montero, J., Alarcon, A., Becker, M. R., & Benson, A. (2023). DSPS: Differentiable stellar population synthesis. *521*(2), 1741–1756. https://doi.org/10.1093/mnras/stad456

Hearin, A. P., Chaves-Montero, J., Becker, M. R., & Alarcon, A. (2021). A Differentiable Model of the Assembly of Individual and Populations of Dark Matter Halos. *The Open Journal of Astrophysics*, *4*(1), 7. https://doi.org/10.21105/astro.2105.05859

Kennedy, J., & Eberhart, R. (1995). Particle swarm optimization. *Proceedings of ICNN'95 - International Conference on Neural Networks*, *4*, 1942–1948 vol.4. https://doi.org/10.1109/ICNN.1995.488968

Li, B., & Wada, K. (2005). Parallelizing particle swarm optimization. *PACRIM. 2005 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing, 2005.*, 288–291. https://doi.org/10.1109/PACRIM.2005.1517282

Scott, D. W. (1992). *Multivariate density estimation: Theory, practice, and visualization*. Wiley. https://doi.org/10.2307/1270280