Rapid Optimization Library: Machine Learning with PyROL

Brian Chen, Denis Ridzal

Trilinos User-Developer Group Meeting

October 22, 2024

LDRD
Laboratory Directed Research and Development

Sandia National Laboratories

U.S. DEPARTMENT OF ENERGY
NNSA National Nuclear Security Administration

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

SAND2024-14401PE

ROL

- ROL (as in *rock and **roll***) is a high-performance Trilinos library for **numerical optimization**.
- Brings an extensive collection of modern optimization algorithms to **any application**.
- The programming interface supports **any computational hardware**, including heterogeneous many-core systems with digital and analog accelerators.
- Used successfully in optimal control, optimal design, inverse problems, image processing and mesh optimization, at **extreme problem scales** (very small and very large).
- Application areas including geophysics, structural dynamics, fluid dynamics, electromagnetics, quantum computing, hypersonics and geospatial imaging.

ROL

RAPID OPTIMIZATION LIBRARY

Numerical optimization made practical:
Any application, any hardware, any problem size.

- Modern optimization algorithms: **inexact, adaptive, stochastic/risk-aware, nonsmooth**.
- Special programming interfaces for simulation-based optimization: **SimOpt**.
- Toolboxes: **OED** for optimal experimental design and **PDE-OPT** for PDE-constrained optimization.

`rol.sandia.gov`

Mathematical Formalism

- ROL solves **smooth nonlinear nonconvex optimization** problems

$$\underset{x}{\text{minimize}} \;\; J(x) \;\; \text{subject to} \;\; \begin{cases} c(x) = 0 \\ \ell \le x \le u \\ Ax = b \end{cases}$$

where $J : \mathcal{X} \to \mathbb{R}$, $c : \mathcal{X} \to \mathcal{C}$ and $A : \mathcal{X} \to \mathcal{D}$, and $\mathcal{X}$, $\mathcal{C}$ and $\mathcal{D}$ are vector spaces.

- ROL additionally solves **stochastic optimization** problems with random inputs ξ,

$$\underset{x}{\text{minimize}} \;\; \mathcal{R}\left[J(x;\xi)\right] \quad \text{etc.} \quad \text{with} \quad J = J(x;\xi) \;\; \text{and} \;\; c = c(x;\xi)$$

where $\mathcal{R}$ is a risk measure, for instance the expectation, $\mathcal{R}\left[J(x;\xi)\right] = \mathbb{E}\left[J(x;\xi)\right]$.

- Finally, ROL solves **nonsmooth optimization** problems

$$\underset{x}{\text{minimize}} \;\; J(x) + \phi(x)$$

where $\phi : \mathcal{X} \to \mathbb{R}$ is nonsmooth and convex, for instance an ℓ_1 regularizer.

A Growing Collection of Algorithms

Type U "Unconstrained"

$$\underset{x}{\text{minimize}} \;\; J(x)$$

$$\text{subject to} \;\; \begin{cases} \\ Ax = b \end{cases}$$

Methods:

- trust region and line search globalization
- gradient descent
- quasi-Newton and inexact Newton
- nonlinear conjugate gradient (CG)
- Cauchy point, dogleg
- Steihaug-Toint truncated CG

Type B "Bound Constrained"

$$\underset{x}{\text{minimize}} \;\; J(x)$$

$$\text{subject to} \;\; \begin{cases} \ell \le x \le u \\ Ax = b \end{cases}$$

Methods:

- projected gradient with line search
- projected Newton with line search
- primal-dual active set
- Lin-Moré trust region
- Kelley-Sachs trust region
- spectral projected gradient (SPG)

Type E "Equality Constrained"

$$\underset{x}{\text{minimize}} \;\; J(x)$$

$$\text{subject to} \;\; \begin{cases} c(x) = 0 \\ \\ Ax = b \end{cases}$$

Methods:

- composite-step sequential quadratic programming (SQP)
- augmented Lagrangian (AL)

Type G "General Constraints"

$$\underset{x}{\text{minimize}} \;\; J(x)$$

$$\text{subject to} \;\; \begin{cases} c(x) = 0 \\ \ell \le x \le u \\ Ax = b \end{cases}$$

Methods:

- AL for equalities and TypeB for bounds
- AL for bounds and TypeE for equalities
- primal interior point
- Moreau-Yosida
- stabilized linearly constrained Lagrangian (LCL)

Type P "Proximable"

$$\underset{x}{\text{minimize}} \;\; J(x) + \phi(x)$$

where
J smooth+nonconvex
ϕ nonsmooth+convex

Methods:

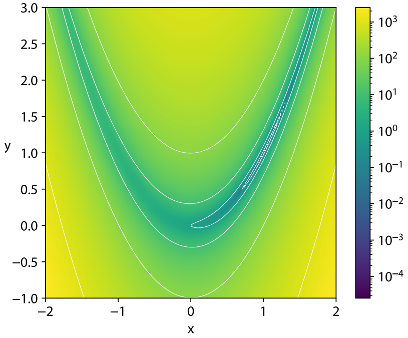
- nonsmooth inexact trust-region methods
- proximal gradient
- spectral proximal gradient
- inexact proximal Newton

Motivating PyROL: Rosenbrock Function

$$f(x_1, \ldots, x_n) = \sum_{i=1}^{N/2} \left[100(x_{2i-1}^2 - x_{2i})^2 + (x_{2i-1} - 1)^2\right]$$

- The Rosenbrock function is a nonconvex function that can be difficult to optimize.
- We highlight ROL's capability to optimize over millions of parameters, $N = 100$ million.
- We will use ROL's **Python** interface. Why?

  Python has had wide reach:
  - efficient and easy-to-learn scripting and programming tool;
  - convenient for teaching and prototyping;
  - "glue code" for large projects;
  - popular, including in AI, e.g., JAX, PyTorch.



Rosenbrock function in $2D$ [1]

[1] https://commons.wikimedia.org/w/index.php?curid=114931732

Optimizing in PyROL

$$\underset{x}{\text{minimize}} \sum_{i=1}^{N/2} \left[100(x_{2i-1}^2 - x_{2i})^2 + (x_{2i-1} - 1)^2\right]$$

```
class RosenbrockObjective(Objective):
    def __init__(self):
        super().__init__()
        self.alpha = 100
    def value(self, x, tol):
        return self.alpha*(x[0]**2-x[1])**2 + (x[0]-1)**2
    def gradient(self, g, x, tol):
        g[0] =  4*self.alpha*(x[0]**2-x[1])*x[0] + 2*(x[0]-1)
        g[1] = -2*self.alpha*(x[0]**2-x[1])
    def hessVec(self, hv, v, x, tol):
        h11 = 12*self.alpha*x[0]**2 - 4*self.alpha*x[1] + 2
        h12 = -4*self.alpha*x[0]
        h22 =  2*self.alpha
        hv[0] = h11*v[0] + h12*v[1]
        hv[1] = h12*v[0] + h22*v[1]
```

```
def main():
    # Set up
    x = NumPyVector(np.array([-3.,-4.]))
    objective = RosenbrockObjective()
    problem = Problem(objective, x)
    problem.check(True)  # Optional
    parameters = ParameterList()
    # Solve
    solver = Solver(problem, parameters)
    solver.solve(getCout())
```

$N = 2$

`ROL::Vector` – A Linear Algebra Interface

- ROL is hardware agnostic.
- You can run ROL on personal computers (in serial and MPI parallel), on GPUs, and on supercomputers by inheriting from `ROL::Vector`.

```python
class NumPyVector(PythonVector):
    # ...
    def axpy(self, alpha, other):
        self.array += alpha*other.array
    def dot(self, other):
        return np.vdot(self.array,
                       other.array)
    def scale(self, alpha):
        self.array *= alpha
```

```python
class TensorVector(PythonVector):
    # ...
    @torch.no_grad()
    def axpy(self, alpha, other):
        self.tensor.add_(other.tensor, alpha=alpha)
    @torch.no_grad()
    def dot(self, other):
        ans = torch.sum(torch.mul(self.tensor,
                                  other.tensor))
        return ans.item()
    @torch.no_grad()
    def scale(self, alpha):
        self.tensor.mul_(alpha)
```

PyTorch Automatic Differentiation

```python
class TorchObjective(Objective):
    def __init__(self):
        super().__init__()
        self.torch_gradient = torch.func.grad(self.torch_value)
    def torch_value(self, x):
        # Returns a scalar torch Tensor
        raise NotImplementedError
    def value(self, x, tol):
        return self.torch_value(x.torch_object).item()
    def gradient(self, g, x, tol):
        ans = self.torch_gradient(x.torch_object)
        g.torch_object = ans
    def hessVec(self, hv, v, x, tol):
        input = torch.func.grad(self.torch_value)
        _, ans = self._forward_over_reverse(input, x.torch_object, v.torch_object)
        hv.torch_object = ans
    def _forward_over_reverse(self, input, x, v):
        # https://github.com/google/jax/blob/main/docs/notebooks/autodiff_cookbook.ipynb
        return torch.func.jvp(input, (x,), (v,))
```

PyTorch Rosenbrock Example

$$\underset{x}{\text{minimize}} \quad \sum_{i=1}^{N/2} \left[100(x_{2i-1}^2 - x_{2i})^2 + (x_{2i-1} - 1)^2\right]$$

```python
class RosenbrockObjective(TorchObjective):
    def torch_value(self, x):
        return torch.sum(100*(x[::2]**2 - x[1::2])**2 + (x[::2] - 1)**2)
```

PyTorch automatically differentiates this problem, and we can also leverage its GPU dispatch:

```python
device = torch.device('cuda')
x = torch.empty(N, requires_grad=False, device=device)
x = TensorVector(x)
```

When $N = 10^8$, the problem takes:
$\sim$ 466 seconds to solve with an Intel Core i9 CPU; and
$\sim$ 7 seconds with an NVIDIA GPU.

Neural Network Example

Set up a neural network in PyTorch.

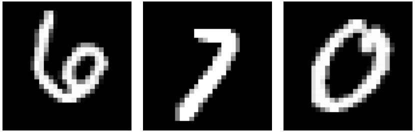
```
class ConvolutionalNet(nn.Module):
    def __init__(self):
        super(ConvolutionalNet, self).__init__()
        self.conv1 = nn.Conv2d(1, 8, 3, 1)
        self.conv2 = nn.Conv2d(8, 8, 3, 1)
        self.fc1 = nn.Linear(1152, 128)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = .....
        output = F.log_softmax(x, dim=1)
        return output
```

Wrap the parameters using `TensorDictVector` and pass the model into a `TorchObjective`.

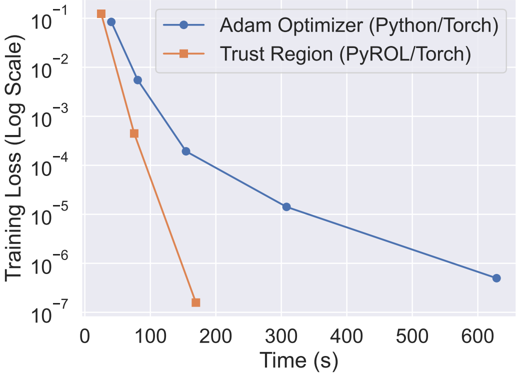
```
def main(data, model, loss_fcn):
    x = TensorDictVector(model.state_dict())
    objective = LeastSquaresObjective(data, model)
    g = TensorDictVector(
        copy.deepcopy(model.state_dict())
    )

    stream = getCout()
    problem = Problem(objective, x, g)
    problem.checkDerivatives(True, stream)

    params = build_parameter_list(iteration_limit)
    solver = Solver(problem, params)
    solver.solve(stream)

    return solver, x
```

Example: MNIST Classification

- Goal: Minimize classification error on MNIST using a convolutional neural network (CNN).
- Optimization Parameters: Around $100,000$ parameters of the CNN.



Images from the MNIST handwritten digits dataset.



MNIST: Comparison of training time and final training errors, between Adam (implemented using PyTorch) versus ROL's trust region (implemented using PyROL and PyTorch).