Exploring Learning to Optimize: End-to-End Approaches for Constrained Optimization and Beyond

Zhonglin Xie

Beijing International Center for Mathematical Research
Peking University

July 18, 2024

Warmup: LISTA

- Least absolute shrinkage and selection operator (LASSO):

$$\min_{x} \frac{1}{2}\|b - Dx\|_2^2 + \lambda\|x\|_1, \text{ where } b = Dx^* + \varepsilon$$

- Iterative Shrinkage Thresholding Algorithm (ISTA):

$$x^{k+1} = \eta_\theta(W_1 b + W_2 x^k), \quad k = 0, 1, 2, \dots$$

where $W_1 = \frac{1}{L}D^\top$, $W_2 = I - \frac{1}{L}D^\top D, \theta = \frac{1}{L}\lambda$

- Learned ISTA (LISTA) with weights $\Theta = \{W_1^k, W_2^k, \theta^k\}_{k=1}^K$:

$$x^{k+1} = \eta_{\theta^k}(W_1^k b + W_2^k x^k), \quad k = 0, 1, \cdots, K-1$$

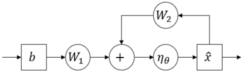


Figure: ISTA

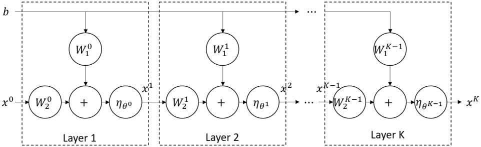


Figure: Unrolled Learned ISTA Network

A Typical Paradigm of Learning to Optimize (L2O)

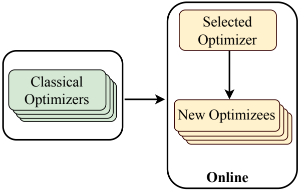
- $\mathcal{F}$: a collection of optimization problems sharing a similar structure

- Learning to Optimize: given x_0 and N

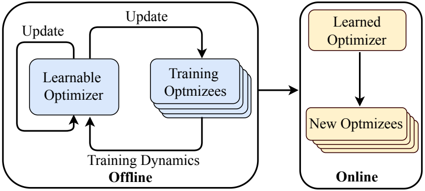
$$\min_{\{\theta_i\}} \quad \mathbb{E}_{f\in\mathcal{F}}\left[\ell_f\left(\{x_i^f\}_{i=0}^N\right)\right]$$

$$\text{s.t.} \quad x_{i+1}^f = x_i^f - \Psi_i(\{x_j^f, \nabla f(x_j^f)\}_{j=0}^i, \theta_i),\ 0 \le i \le N-1$$

- ℓ_f emphasizes the dependence on f. The symbols Ψ_i represent neural networks, while θ_i refers to their corresponding weights



(a) Classic Optimizer



(b) Learning to Optimize

The Bitter Lesson of Model-based L2O

> *The biggest lesson that can be read from 70 years of AI research is that general methods that leverage computation are ultimately the most effective, and by a large margin.*
> *—— The Bitter Lesson, March 13, 2019, Rich Sutton.*

- Model-based L2O does not utilize the power of computation
- It needs a problem-specific customization, not general enough
- Can we learn a direct mapping from data to solutions

Deep Constraint Completion and Correction (DC3)

- Given the problem data x

  $$\min_{y \in \mathbb{R}^n} \; f_x(y) \quad \text{s.t. } g_x(y) \leq 0, h_x(y) = 0$$

  where f, g, and h are potentially nonlinear and non-convex

- Training a neural network N_θ to approximate y given x

- A naive **soft loss**:

  $$\ell_{\text{soft}}(\hat{y}) = f_x(\hat{y}) + \lambda_g \left\| \text{ReLU}\left(g_x(\hat{y})\right) \right\|_2^2 + \lambda_h \left\| h_x(\hat{y}) \right\|_2^2, \quad \hat{y} = N_\theta(x)$$

- **Supervised learning** framework on example (x, y):

  $$\ell(\hat{y}) = \|\hat{y} - y\|_2^2, \quad \hat{y} = N_\theta(x)$$

- Both can lead in practice to **highly infeasible** outputs

DC3 equality completion

- Enforcing equality constraints by **variable elimination**

- First output a subset of the variables, z, then infer the remaining, $\varphi_x(z)$, according to $h_x([z, \varphi_x(z)]) = 0$

- **Backpropagation**:

$$0 = \frac{\mathrm{d}}{\mathrm{d}z} h_x\left(\begin{bmatrix} z \\ \varphi_x(z) \end{bmatrix}\right) = \frac{\partial h_x}{\partial z} + \frac{\partial h_x}{\partial \varphi_x(z)} \frac{\partial \varphi_x(z)}{\partial z} = J_z^h + J_\varphi^h \frac{\partial \varphi_x(z)}{\partial z}$$

$$\Rightarrow \partial \varphi_x(z)/\partial z = -\left(J_\varphi^h\right)^{-1} J_z^h$$

- Backpropagate losses through the network:

$$\frac{\mathrm{d}\ell}{\mathrm{d}z} = \frac{\partial \ell}{\partial z} + \underbrace{\frac{\partial \ell}{\partial \varphi_x(z)} \frac{\partial \varphi_x(z)}{\partial z}}_{\text{left matrix-vector product}} = \frac{\partial \ell}{\partial z} - \frac{\partial \ell}{\partial \varphi_x(z)} \left(J_\varphi^h\right)^{-1} J_z^h$$

Inequality correction (Highly questionable)

- Decreasing the inequality violation by taking a **gradient step**
- Denote the gradient of inequality violation w.r.t. $[z, \varphi_x(z)]$ as

$$\Delta z = \nabla_z \left\| \mathrm{ReLU}\left(g_x \left(\begin{bmatrix} z \\ \varphi_x(z) \end{bmatrix} \right) \right) \right\|_2^2, \quad \Delta \varphi_x(z) = \frac{\partial \varphi_x(z)}{\partial z} \Delta z$$

- For a step size $\gamma > 0$, we define:

$$\rho_x \left(\begin{bmatrix} z \\ \varphi_x(z) \end{bmatrix} \right) = \begin{bmatrix} z - \gamma \Delta z \\ \varphi_x(z) - \gamma \Delta \varphi_x(z) \end{bmatrix}, \quad \rho_x^{(t)} = \underbrace{\rho_x \circ \cdots \circ \rho_x}_{t \text{ times}}$$

Algorithm: Deep Constraint Completion and Correction

Algorithm Deep Constraint Completion and Correction (DC3)

```
Require: Assume equality completion procedure $\varphi_x : \mathbb{R}^m \to \mathbb{R}^{n-m}$
procedure TRAIN($X$)
    init neural network $N_\theta : \mathbb{R}^d \to \mathbb{R}^m$
    while not converged do
        for $x \in X$ do
            compute partial set of variables $z = N_\theta(x)$
            complete to full set of variables $\tilde{y} = \begin{bmatrix} z^\top & \varphi_x(z)^\top \end{bmatrix}^\top \in \mathbb{R}^n$
            correct to feasible (or approx. feasible) solution $\hat{y} = \rho_x^{(t_{\text{train}})}(\tilde{y})$
            compute constraint-regularized loss $\ell_{\text{soft}}(\hat{y})$
            update $\theta$ using $\nabla_\theta \ell_{\text{soft}}(\hat{y})$
        end for
    end while
end procedure
procedure TEST($x, N_\theta$)
    compute partial set of variables $z = N_\theta(x)$
    complete to full set of variables $\tilde{y} = \begin{bmatrix} z^\top & \varphi_x(z)^\top \end{bmatrix}^\top$
    correct to feasible solution $\hat{y} = \rho_x^{(t_{\text{test}})}(\tilde{y})$
    return $\hat{y}$
end procedure
```

Comments on DC3

- Intuitive and effective for enforcing feasibility
- Assuming $[z, \varphi_x(z)]$ **already be close to feasible** before correction
- The correction process is proved to converge for **linear constraints**
- The obtained **feasible** solution may be **sub-optimal**
- Backpropagating through $\rho_x^{(t_{\text{train}})}(\tilde{y})$ needs more justification

Lagrangian Duality for Constrained Deep Learning

- Consider the parametric constrained optimization

$$\mathcal{O}(d) = \underset{y}{\operatorname{argmin}}\ f(y, d) \quad \text{subject to } g_i(y, d) \leqslant 0 \quad (\forall i \in [m])$$

- Given a set of samples $D = \{(d_l, y_l = \mathcal{O}(d_l))\}_{l=1}^{n}$, we solve

$$\theta^* = \underset{\theta}{\operatorname{argmin}} \sum_{l=1}^{n} \mathcal{L}\left(N_\theta\left(d_l\right), y_l\right)$$
$$\text{subject to } g_i\left(N_\theta\left(d_l\right), d_l\right) \leqslant 0 \quad (\forall i \in [m], l \in [n])$$

where $\mathcal{L}$ is a loss function, N_{θ^*} is the learned optimizer

Lagrangian Dual Framework for Constrained Optimization

- Given multipliers $\lambda = (\lambda_1, \ldots, \lambda_m)$, **Lagrangian loss** writes

$$\mathcal{L}_\lambda \left(N_\theta \left(d_l \right), y_l, d_l \right) = \mathcal{L} \left(N_\theta \left(d_l \right), y_l \right) + \sum_{i=1}^{m} \lambda_i g_i \left(N_\theta \left(d_l \right), d_l \right)$$

- $N_{\theta^*(\lambda)}$ is an approximation of the oracle $\mathcal{O}$ with

$$\theta^*(\lambda) = \operatorname*{argmin}_\theta \sum_{l=1}^{n} \mathcal{L}_\lambda \left(N_\theta \left(d_l \right), y_l, d_l \right)$$

- The Lagrangian dual computes the optimal multipliers

$$\lambda^* = \operatorname*{argmax}_\lambda \min_\theta \sum_{l=1}^{n} \mathcal{L}_\lambda \left(N_\theta \left(d_l \right), y_l, d_l \right)$$

- The strongest Lagrangian relaxation of $\mathcal{O}$ is $N_{\theta^*(\lambda^*)}$

Algorithm

Algorithm LDF for Constrained Optimization Problems

```
Require: $D = (d_l, y_l)_{l=1}^n$ : Training data
$\alpha, s = (s_0, s_1, \dots)$ : Optimizer and Lagrangian step sizes
$\lambda_i^0 \leftarrow 0 \quad \forall i \in [m]$
for epoch $k = 0, 1, \dots$ do
    for all $(y_l, d_l) \in D$ do
        $\hat{y}_l \leftarrow N_{\theta(\lambda^k)}(d_l)$
        $\theta \leftarrow \theta - \alpha \nabla_\theta \mathcal{L}_{\lambda^k}(\hat{y}_l, y_l, d_l)$
    end for
    $\lambda_i^{k+1} \leftarrow \lambda_i^k + s_k \sum_{l=1}^n g_i(\hat{y}_l, d_l) \quad \forall i \in [m]$
end for
```

Self-Supervised Primal-Dual Learning

- Lagrangian dual framework has no guarantee for feasibility!
- Consider the **Augmented Lagrangian loss**

$$\mathcal{L}_{\lambda}\left(\hat{y}, d_l\right) = f\left(\hat{y}, d_l\right) + \sum_{i=1}^{m} \lambda_i g_i\left(\hat{y}, d_l\right) + \rho \sum_{i=1}^{m} \nu(g_i\left(\hat{y}, d_l\right))$$

  where $\hat{y} = N_{\theta}\left(d_l\right)$ and $\nu(\cdot) = \max\{\cdot, 0\}^2$ measures the violation

- Dual learning uses the dual network M_{ϕ} to obtain λ^*

Algorithm

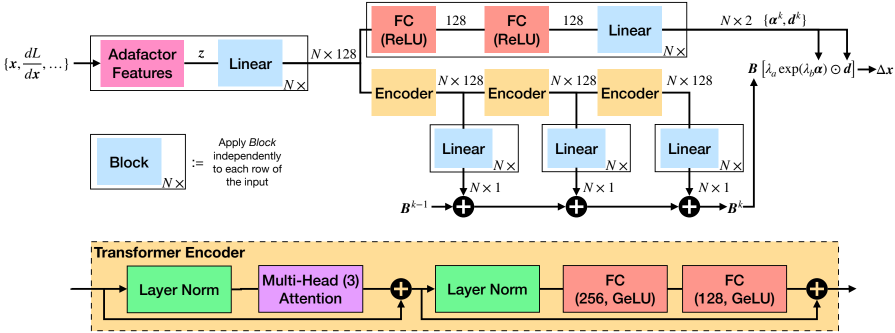
Algorithm Self-Supervised Primal-Dual Learning

```
Require: $D = (d_l, y_l)_{l=1}^{n}$ : Training data
$\alpha, \beta, \rho_{\max}$ : Primal and dual step sizes, upper bound of $\rho$
$\lambda_i^0 \leftarrow 0 \quad \forall i \in [m]$
for epoch $k = 0, 1, \ldots$ do
    for all $(y_l, d_l) \in D$ do
        $\hat{y}_l \leftarrow N_{\theta^k}(d_l)$
        $\theta \leftarrow \theta - \alpha \nabla_\theta \mathcal{L}_{\lambda^k}(\hat{y}_l, d_l)$
    end for
    for all $(y_l, d_l) \in D$ do
        Freeze $\lambda^k \leftarrow M_{\phi^k}(d_l), \hat{y}_l \leftarrow N_{\theta^k}(d_l)$
        $\phi \leftarrow \phi - \beta \nabla_\phi \| M_\phi(d_l) - \max\{\lambda^k + \rho g(\hat{y}_l), 0\} \|$
    end for
    $\rho \leftarrow \min\{\alpha\rho, \rho_{\max}\}$
end for
```

Transformer-based L2O

- JAX learned optimization package
- Inspired from BFGS, it constructs a rank one update each step

$$\Delta \mathbf{x}^k = \mathbf{B}^k \mathbf{s}^k, \quad \tilde{\mathbf{B}}^{k+1} = \mathbf{B}^k + \sum_{l=1}^{L} \mathbf{u}_l^k \left(\mathbf{u}_l^k\right)^\top, \quad \mathbf{B}^{k+1} = \tilde{\mathbf{B}}^{k+1} / \left\|\tilde{\mathbf{B}}^{k+1}\right\|$$



VeLO: Training Versatile Learned Optimizers by Scaling Up

- JAX learned optimization package
- Trained with **four thousand TPU-months** of compute
- Requires no hyperparameter tuning, automatically adapting

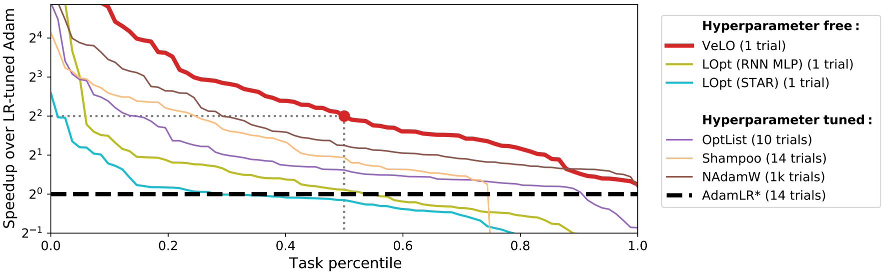
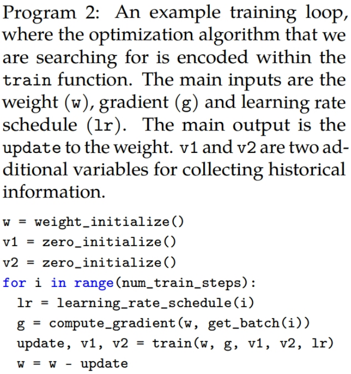


Figure: Optimizer performance on the 83 canonical tasks in the VeLOdrome

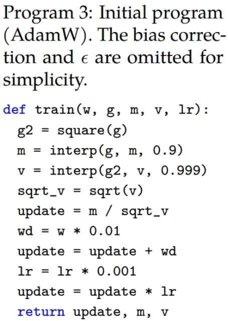
Symbolic Discovery of Optimization Algorithms

- Google automl repository
- A total cost of 3K TPU V2 days
- Discover the **Lion** (EvoL**ved** **Si**gn M**o**me**n**tum) algorithm



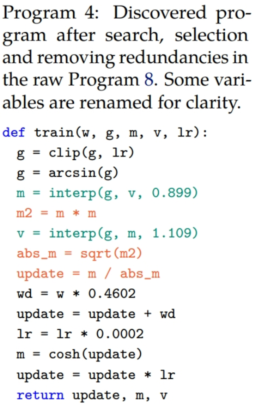
Program 2: An example training loop, where the optimization algorithm that we are searching for is encoded within the train function. The main inputs are the weight (w), gradient (g) and learning rate schedule (lr). The main output is the update to the weight. v1 and v2 are two additional variables for collecting historical information.

```
w = weight_initialize()
v1 = zero_initialize()
v2 = zero_initialize()
for i in range(num_train_steps):
  lr = learning_rate_schedule(i)
  g = compute_gradient(w, get_batch(i))
  update, v1, v2 = train(w, g, v1, v2, lr)
  w = w - update
```



Program 3: Initial program (AdamW). The bias correction and ϵ are omitted for simplicity.

```
def train(w, g, m, v, lr):
  g2 = square(g)
  m = interp(g, m, 0.9)
  v = interp(g2, v, 0.999)
  sqrt_v = sqrt(v)
  update = m / sqrt_v
  wd = w * 0.01
  update = update + wd
  lr = lr * 0.001
  update = update * lr
  return update, m, v
```



Program 4: Discovered program after search, selection and removing redundancies in the raw Program 8. Some variables are renamed for clarity.

```
def train(w, g, m, v, lr):
  g = clip(g, lr)
  g = arcsin(g)
  m = interp(g, v, 0.899)
  m2 = m * m
  v = interp(g, m, 1.109)
  abs_m = sqrt(m2)
  update = m / abs_m
  wd = w * 0.4602
  update = update + wd
  lr = lr * 0.0002
  m = cosh(update)
  update = update * lr
  return update, m, v
```

Many Thanks For Your Attention!