Supplemental Material: ∇-Prox: Differentiable Proximal Algorithm Modeling for Large-Scale Optimization

ZEQIANG LAI*, Beijing Institute of Technology, China
KAIXUAN WEI*, Princeton University, USA and McGill University, Canada
YING FU, Beijing Institute of Technology, China
PHILIPP HÄRTEL, Fraunhofer IEE, Germany and Princeton University, USA
FELIX HEIDE, Princeton University, USA

ACM Reference Format:
Zeqiang Lai, Kaixuan Wei, Ying Fu, Philipp Härtel, and Felix Heide. 2023. Supplemental Material: ∇-Prox: Differentiable Proximal Algorithm Modeling for Large-Scale Optimization. *ACM Trans. Graph.* 42, 4, Article 105 (July 2023), 32 pages. https://doi.org/10.1145/3592144

OUTLINE

This supplementary material provides further details and results to support the content of the main paper. It is organized as follows.

Section A: We provide more details for applying deep equilibrium learning to proximal algorithms, including the conversion from proximal optimization to fixed-point iteration and its gradient derivation.

Section B: In the interest of clarity, we introduce algorithm principles and implementation details of learning proximal solvers with deep reinforcement learning.

Section C: We present more details about the solver construction pipeline with examples of how eliminating calculations enables more efficient forward/backward evaluations.

Section D: To showcase the ease of use and scalability of ∇-Prox, we sketch a step-by-step tutorial about adding customized operators and proximal algorithms in light of users' interest. A sanity check mechanism is also introduced to ensure implementation accuracy.

Section E: With the purpose of completeness, we provide further details on the implementation of ∇-Prox, including its built-in linear operators, proximal functions, proximal algorithms, as well as the trainer interface to optimize solvers.

Section F: To help users familiarize ∇-Prox language and its usage, we provide examples with accompanying code explained in a line-by-line fashion. More user-friendly tutorials are available in our released codebase.

Section G: To reach a broader audience of the graphics community, we present general introductory background of integrated energy system planning problems. The convergence loss mentioned in Section 5.4 of the main paper is also introduced here.

Section H: We provide further details (*e.g.*, experimental setup, training script) on the experimental results for a wide range of applications.

*indicates equal contribution.

2023. 0730-0301/2023/7-ART105 $15.00
https://doi.org/10.1145/3592144

A ADDITIONAL DETAILS ON DEEP EQUILIBRIUM LEARNING

In this section, we first provide a general review of deep equilibrium learning (DEQ), then we detail how we transform the proximal optimization routines into a fixed-point iteration on which DEQ operates, and finally, we show how to compute the gradients that backpropagate through whole the proximal optimization trajectory efficiently by leveraging DEQ.

A.1 The Overview of Deep Equilibrium Learning

The DEQ was originally applied to infinite-depth networks for sequential data, which is an area orthogonal to optimization [Bai et al. 2019]. In a sense, DEQ is a direct solver for approaches that repeatedly apply deep sequence models, *e.g.*, RNN and LSTM, to converge towards some fixed points. To this end, DEQ starts with a reformulation of the deep sequence model into a fixed-point iteration

$$\mathbf{x}^{\infty} = f_{\theta}\left(\mathbf{x}^{\infty}; \mathbf{y}\right), \tag{1}$$

where f_{θ} denotes the deep sequence model, $\mathbf{x}$ represents to the hidden states of the model, and $\mathbf{y}$ denotes the input to the model. Instead of repeatedly evaluating until convergence, DEQ directly finds the equilibrium points via root-finding, which can be conceptually treated as an infinite iteration. One of the most prominent advantages of DEQ is that the backward gradient can be analytically obtained using implicit differentiation with constant memory cost.

A.2 Formulating Proximal Optimization as Fixed-Point Iteration

To incorporate DEQ into ∇-Prox, we have to convert the proximal optimization into the computing format of DEQ, *i.e.*, in the form of fixed-point iteration. Every proximal algorithm is an optimization method that finds the optimal solution by iterating over a series of variable updates. For example, the updates for ADMM algorithms consist of two primal updates for the main variable $\mathbf{x}$ and the auxiliary variable $\mathbf{v}$ and one dual update for the multiplier $\mathbf{u}$. To convert any proximal algorithm into a fixed-point iteration, we adopt the following steps.

(1) Decouple the dependency of the optimization variables of the same step. For example, the original ADMM typically uses the latest version of the optimization variable to update the others, *e.g.*, $\mathbf{u}^t = g(\mathbf{x}^t, \mathbf{v}^{t-1})$. To enable the conversion, we have to change it to $\mathbf{u}^t = g(\mathbf{x}^{t-1}, \mathbf{v}^{t-1})$ where every optimization variable only depends on the variables from the last step.
(2) Stack all the optimization variables into a single input. For example, we stack the variables $\mathbf{x}, \mathbf{v}, \mathbf{u}$ of ADMM into a single one $\mathbf{X}$. This helps to build a unified interface for different proximal algorithms with a different number of optimization variables.
(3) Stack all the updates of the chosen proximal algorithm into a single one. Taking ADMM as an example, we stack the $\mathbf{x}$ update, $\mathbf{v}$ update, and $\mathbf{u}$ update into a single one $\mathbf{X}^t = f(\mathbf{X}^{t-1})$ that takes stacked variables from the previous step X^{t-1} as input and outputs the stacked variables at the current step $\mathbf{X}^t$.

In summary, the fixed point iteration of proximal optimization can be formulated as

$$\mathbf{X}^{\infty} = f_{\theta}\left(\mathbf{X}^{\infty}; \mathbf{M}\right), \tag{2}$$

where $\mathbf{X}^{\infty}$ is a compound variable stacking all optimization variables, *e.g.*, $\mathbf{x}, \mathbf{v}, \mathbf{u}$ for ADMM, f_{θ} represents a single round of updates of optimization variables $\mathbf{x}, \mathbf{v}, \mathbf{u}$, and $\mathbf{M}$ denotes any other input for the updates, *e.g.*, hyperparameters ρ and observation $\mathbf{y}$.

A.3 Detailed Derivation of Gradient Calculation of DEQ

With the fixed-point iteration formulation of proximal optimization, we could derive an efficient gradient calculation method using implicit differentiation for fast parameter tuning. In a nutshell, consider the gradient of

the parameters θ with respect to some scalar loss $\mathcal{L}$, we first apply the chain rule as

$$\frac{\partial \mathcal{L}}{\partial \theta} = (\frac{\partial \mathbf{X}^{\infty}}{\partial \theta})^{T} \frac{\partial \mathcal{L}}{\partial \mathbf{X}^{\infty}}, \tag{3}$$

where the second term $\frac{\partial \mathcal{L}}{\partial \mathbf{X}^{\infty}}$ can usually be automatically computed by auto-diff system. To efficiently obtain the first term $\frac{\partial \mathbf{X}^{\infty}}{\partial \theta}$, we apply the chain rule to obtain

$$\frac{\partial \mathbf{X}^{\infty}}{\partial \theta} = \frac{\partial \mathbf{X}^{\infty}}{\partial f_{\theta}\left(\mathbf{X}^{\infty}; \mathbf{M}\right)} \frac{\partial f_{\theta}\left(\mathbf{X}^{\infty}; \mathbf{M}\right)}{\partial \theta}. \tag{4}$$

The first term of (4) can be derived by differentiate both size of (2) again,

$$\frac{\partial \mathbf{X}^{\infty}}{\partial f_{\theta}\left(\mathbf{X}^{\infty}; \mathbf{M}\right)} = \frac{\partial f_{\theta}\left(\mathbf{X}^{\infty}; \mathbf{M}\right)}{\partial \mathbf{X}^{\infty}} \frac{\partial \mathbf{X}^{\infty}}{\partial f_{\theta}\left(\mathbf{X}^{\infty}; \mathbf{M}\right)}, \tag{5}$$

By rearranging the items,

$$\left(\mathbf{I} - \frac{\partial f_{\theta}\left(\mathbf{X}^{\infty}; \mathbf{M}\right)}{\partial \mathbf{X}^{\infty}}\right) \frac{\partial \mathbf{X}^{\infty}}{\partial f_{\theta}\left(\mathbf{X}^{\infty}; \mathbf{M}\right)} = \mathbf{I}, \tag{6}$$

we could obtain,

$$\frac{\partial \mathbf{X}^{\infty}}{\partial f_{\theta}\left(\mathbf{X}^{\infty}; \mathbf{M}\right)} = \left(\mathbf{I} - \frac{\partial f_{\theta}\left(\mathbf{X}^{\infty}; \mathbf{M}\right)}{\partial \mathbf{X}^{\infty}}\right)^{-1}. \tag{7}$$

This can be plugged back to (4) to obtain

$$\frac{\partial \mathbf{X}^{\infty}}{\partial \theta} = \left(\mathbf{I} - \frac{\partial f_{\theta}(\mathbf{X}^{\infty}; \mathbf{M})}{\partial \mathbf{X}^{\infty}}\right)^{-1} \frac{\partial f_{\theta}\left(\mathbf{X}^{\infty}; \mathbf{M}\right)}{\partial \theta}. \tag{8}$$

Then, (8) can be plugged back into (3) to obtain

$$\frac{\partial \mathcal{L}}{\partial \theta} = \underbrace{\left(\frac{\partial f_{\theta}\left(\mathbf{X}^{\infty}; \mathbf{M}\right)}{\partial \theta}\right)^{T}}_{\text{Single-step Gradient}} \underbrace{\left(\mathbf{I} - \frac{\partial f_{\theta}(\mathbf{X}^{\infty}; \mathbf{M})}{\partial \mathbf{x}^{\infty}}\right)^{-T} \frac{\partial \mathcal{L}}{\partial \mathbf{X}^{\infty}}}_{\text{Inverse Jacobian-vector}}. \tag{9}$$

The first part is a single-step gradient that can be computed by utilizing auto-diff. The second part is an inverse Jacobian-vector product that can be obtained with another fixed-point problem. To illustrate, we first define the second part as β^{∞}

$$\beta^{\infty} = \left(\mathbf{I} - \frac{\partial f_{\theta}(\mathbf{X}^{\infty}; \mathbf{M})}{\partial \mathbf{x}^{\infty}}\right)^{-T} \frac{\partial \mathcal{L}}{\partial \mathbf{X}^{\infty}}. \tag{10}$$

This can be transformed to

$$\beta^{\infty} = \left(\frac{\partial f_{\theta}(\mathbf{X}^{\infty}; \mathbf{M})}{\partial \mathbf{X}^{\infty}}\right)^{T} \beta^{\infty} + \frac{\partial \mathcal{L}}{\partial \mathbf{X}^{\infty}}, \tag{11}$$

which can be treated as a fixed-point problem.

In summary, we see that the $\frac{\partial \mathcal{L}}{\partial \theta}$ can now be calculated by solving a single-step gradient and a fixed-point problem. The fixed-point problem is addressed on the fly without requiring intermediate-state storage. Therefore, the memory cost can be substantially reduced to the amount of one-step optimization.

B LEARNING SOLVERS VIA REINFORCEMENT LEARNING

In this section, we elaborate on the reinforcement-learning-based strategy for learning the automatic parameter tuner within a proximal solver introduced in Section 4.5 of the main paper. First, we describe how to formulate the automated parameter selection of proximal algorithms in a reinforcement learning (RL) context. Then, we introduce the RL algorithm that optimizes a policy network for automatic parameter tuning. Interested readers are also referred to [Wei et al. 2022] for additional technical detail.

B.1 Formulation Automated Parameter Selection as an RL Problem

Our goal is to automatically select a sequence of internal algorithm parameters, *e.g.*, $(\rho^0, \lambda^0, \rho^1, \lambda^1, \cdots, \rho^\tau, \lambda^\tau)$ to guide the optimization (for instance, Algorithm 1 in the main paper) for problem-specific objectives of interest (for example in image optimization, the ultimate objective is to recover an image that closes to the underlying ground truth, meanwhile speeding up the convergence). This problem can be formulated as a Markov decision process (MDP) addressed via reinforcement learning. Reinforcement learning is a subarea of machine learning related to how an agent should act within an environment, to maximize its cumulative rewards [François-Lavet et al. 2018]. We briefly introduce basic concepts from RL, and how we formulate our automated parameter selection problem as an RL problem.

We denote the MDP by the tuple $(\mathcal{S}, \mathcal{A}, p, r)$, where $\mathcal{S}$ is the state space, $\mathcal{A}$ is the action space, p is the transition function related to environmental dynamics, and r is the reward function. Take the image optimization task as an example, $\mathcal{S}$ is the space of optimization (primal and dual) variable states, which includes the initialization (x^0, v^0, u^0) and all intermediate results (x^k, v^k, u^k) in the optimization process. $\mathcal{A}$ is the space of internal parameters, including both discrete termination time τ and the continuous penalty/weighting parameters (ρ^k, λ^k). The transition function $p : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$ maps input state $s \in \mathcal{S}$ to its outcome state $s' \in \mathcal{S}$ after taking action $a \in \mathcal{A}$. The state transition can be expressed as $s^{t+1} = p(s^t, a^t)$, which is composed of one or several iterations in the optimization. On each transition, the environment emits a reward with respect to the reward function $r : S \times \mathcal{A} \to \mathbb{R}$, which evaluates actions given the state. Applying a sequence of parameters to the initial state s_0 results in a trajectory T of states, actions and rewards, *i.e.* $T = \{s_0, a_0, r_0, \cdots, s_N, a_N, r_N\}$. Then, we define the return R_t of a trajectory T as the summation of discounted rewards after s_t:

$$R_t = \sum_{t'=0}^{N-t} \gamma^{t'} r(s_{t+t'}, a_{t+t'}), \tag{12}$$

where $\gamma \in [0, 1)$ is a discount factor and prioritizes earlier rewards over later ones.

Our objective is to learn a policy π, denoted as $\pi(a|s) : \mathcal{S} \to \mathcal{A}$ for the decision-making agent, in order to maximize the objective defined as

$$J(\pi) = \mathbb{E}_{s_0 \sim S_0, T \sim \pi}[R_0], \tag{13}$$

where $\mathbb{E}$ represents expectation, s_0 is the initial state, and S_0 is the corresponding initial state distribution. Intuitively, the objective describes the expected return over all possible trajectories induced by the policy π. The expected return on states and state-action pairs under the policy π are defined by state-value functions V^π and action-value functions Q^π respectively, *i.e.*

$$V^\pi(s) = \mathbb{E}_{T \sim \pi}[R_0 | s_0 = s], \tag{14}$$

$$Q^\pi(s, a) = \mathbb{E}_{T \sim \pi}[R_0 | s_0 = s, a_0 = a]. \tag{15}$$

In our task, we divide actions into two components, *i.e.* $a = (a_1, a_2)$[1], which includes a discrete decision a_1 on termination time τ and a continuous decision a_2 on penalty/weighting parameter (ρ, λ). The policy also has two

[1]Strictly speaking, $a_t = (a_{t1}, a_{t2})$. Here, we omit the notation t (time step) for simplicity.

sub-policies: $\pi = (\pi_1, \pi_2)$, a stochastic policy and a deterministic policy, which generate a_1 and a_2 respectively. The role of π_1 is to determine when to end the iterative algorithm by sampling a boolean-valued outcome a_1 from a two-class categorical distribution $\pi_1(\cdot|s)$, which is calculated based on the current state s. If $a_1 = 0$, the next iteration will proceed; otherwise, the optimization terminates and outputs the final state. In contrast to the stochastic policy π_1, π_2 is treated deterministically, *i.e.*, $a_2 = \pi_2(s)$. Because π_2 is differentiable with respect to the environment, its gradient can be accurately estimated.

In RL, the environment is characterized by two components, *i.e.* the environment dynamics and reward function. The dynamics of the environment are represented by the transition function p, which is associated with the differentiable proximal algorithm in our task. The transition function p in this scenario involves m iterations of optimization at each time step. The value of m determines the level of control over the termination time of the optimization process, with larger values leading to coarser control but more efficient decision-making. To avoid infinite optimization loops, the maximum time step N is imposed, leading to $m \times N$ iterations of the optimization at most. To take both algorithm effectiveness and runtime into account, the reward function is defined as:

$$r(s_t, a_t) = [\zeta(p(s_t, a_t)) - \zeta(s_t)] - \eta, \tag{16}$$

The first term, $\zeta(p(s_t, a_t)) - \zeta(s_t)$, measures the performance improvement (*e.g.*, PSNR in image recovery) made by the policy, where $\zeta(\cdot)$ is the performance metric of the task at hand. The second term, η, penalizes the policy for not terminating at step t, with η determining the degree of penalty. A negative reward is given if the performance gain does not exceed the degree of penalty, encouraging the policy to stop the iteration when the gain diminishes.

B.2 Optimizing Parameter Selection Policy

Next, we introduce a mixed algorithm for learning a parameter selection policy that combines both model-free and model-based reinforcement learning. Specifically, model-free RL (agnostic to the environment dynamics) is used to train π_1, while model-based RL is utilized to optimize π_2 to make full use of the environment model. We employ the actor-critic framework [Sutton et al. 2000], which includes a policy network, $\pi_\theta(a_t|s_t)$ (the actor), and a value network, $V^\pi_\phi(s_t)$ (the critic), to formulate the policy and state-value function, respectively.

The policy and value networks are designed to be simple yet effective. We use residual structures (ResNet-18) [He et al. 2016] as the feature extractors in both networks, followed by fully-connected layers and activation functions to produce the desired outputs. It is worth noting that the additional computation cost of the policy network is minimal compared to the iteration cost of proximal algorithms.

The policy and value networks are learned in an alternating manner. For each gradient step, we update the parameters of the value network, ϕ, by minimizing the loss function:

$$L_\phi = \mathbb{E}_{s\sim B, a\sim\pi_\theta(s)}\left[\frac{1}{2}(r(s,a) + \gamma V^\pi_{\hat{\phi}}(p(s,a)) - V^\pi_\phi(s))^2\right], \tag{17}$$

where B is the distribution of previously sampled states, which is implemented by a state buffer, and serves as a form of experience replay mechanism [Lin 1992]. This is observed to "smooth" the training data distribution [Mnih et al. 2013]. The update uses a target value network $V^\pi_{\hat{\phi}}$ to stabilize the training process [Mnih et al. 2015], where $\hat{\phi}$ is the exponential moving average of the value network weights.

The policy network is composed of two sub-policies that share convolutional layers to extract image features. These are then followed by two separate groups of fully-connected layers that produce the termination probability, $\pi_1(\cdot|s)$, after softmax, or the penalty/balancing parameters, $\pi_2(s)$, after sigmoid. The parameters of the sub-policies are denoted as θ_1 and θ_2, respectively, and the goal is to optimize $\theta = (\theta_1, \theta_2)$ so that the objective $J(\pi_\theta)$ is maximized. The policy network is trained using policy gradient methods [Peters and Schaal 2006]. The gradient of θ_1 is estimated by a likelihood estimator in a model-free manner, while the gradient of θ_2 is estimated relying

Algorithm 1 Parameter Selection Policy Training Scheme

Require: Image dataset D, degradation operator $g(\cdot)$, learning rates l_θ, l_ϕ, weight parameter β.

Initialize network parameters θ, ϕ, $\hat{\phi}$ and state buffer B.
for each training iteration **do**
  sample initial state s_0 from D via $g(\cdot)$
  for environment step $t \in [0, N)$ **do**
    $a_t \sim \pi_\theta(a_t|s_t)$
    $s_{t+1} \sim p(s_{t+1}|s_t, a_t)$
    $B \leftarrow B \cup \{s_{t+1}\}$
    break if the boolean outcome of a_t equals to 1
  end for
  for each gradient step **do**
    sample states from the state buffer B
    $\theta_1 \leftarrow \theta_1 + l_\theta \nabla_{\theta_1} J(\pi_\theta)$
    $\theta_2 \leftarrow \theta_2 + l_\theta \nabla_{\theta_2} J(\pi_\theta)$
    $\phi \leftarrow \phi - l_\phi \nabla_\phi L_\phi$
    $\hat{\phi} \leftarrow \beta\phi + (1-\beta)\hat{\phi}$
  end for
end for

Ensure: Learned policy network π_θ

on backpropagation via the environment dynamics in a model-based manner. Specifically, for discrete termination time decision π_1, we apply the policy gradient theorem [Sutton et al. 2000] to obtain unbiased the Monte Carlo estimate of $\nabla_{\theta_1} J(\pi_\theta)$ using the advantage function $A^\pi(s,a) = Q^\pi(s,a) - V^\pi(s)$ as the target, which is computed as

$$\nabla_{\theta_1} J(\pi_\theta) = \mathbb{E}_{s\sim B, a\sim\pi_\theta(s)} \left[\nabla_{\theta_1} \log \pi_1(a_1|s)\, A^\pi(s,a) \right]. \tag{18}$$

For continuous denoising strength and penalty parameter selection π_2, we utilize the deterministic policy gradient theorem [Silver et al. 2014] to formulate its gradient that reads:

$$\nabla_{\theta_2} J(\pi_\theta) = \mathbb{E}_{s\sim B, a\sim\pi_\theta(s)} \left[\nabla_{a_2} Q^\pi(s,a) \nabla_{\theta_2} \pi_2(s) \right], \tag{19}$$

where we approximate the action-value function $Q^\pi(s,a)$ by $r(s,a) + \gamma V_\phi^\pi(p(s,a))$ given its unfolded definition [Sutton and Barto 2018].

By using the chain rule, we can directly calculate the gradient of θ_2 through backpropagation with the reward function, the value network, and the transition function. This is in contrast to relying solely on the gradient backpropagated from the learned action-value function in the model-free DDPG algorithm [Lillicrap et al. 2016].

The training algorithm for our policy learning is outlined in Algorithm 1. It requires an image dataset D, a degradation operator $g(\cdot)$, learning rates l_θ, l_ϕ, and a weight parameter β. To generate the initial states s_0, we define the degradation operator $g(\cdot)$ as a combination of a forward model and an initialization function. The forward model maps the underlying image x to its observation y, while the initialization function generates the initial estimate x_0 from the observation y. For linear inverse problems, $g(\cdot)$ is typically defined as the composition of the forward operator and the adjoint operator of the problem (for example, $g(\cdot)$ is the composition of the partially-sampled Fourier transform and the inverse Fourier transform in CS-MRI).

C ADDITIONAL DETAILS ON SOLVER CONSTRUCTION

C.1 Solver Construction Pipeline

The solver construction/compilation pipeline is shown in Figure I. We here provide a brief introduction to the different compilation stages and we refer interested readers to ProxImaL [Heide et al. 2016] for more details from which it mostly inherits.

Problem Transformation. ∇-Prox supports compositing different proxable penalty functions, whose corresponding linear operators can be composed as well. These compositions produce a directed acyclic graph (DAG) that is similar to the abstract syntax tree (AST) [Wile 1997] of a conventional computer program. The problem transformation plays the same role as the AST transformation [Kruse 2021] phase of the compilation of conventional programming language. This typically involves recognizing patterns on the proximal and linear operators in the graph and performing a series of sub-graph rewriting or reduction. For example, constant folding is performed to recognize and evaluate the constant expression in the problem DAG (*e.g.*, extracting the offset of the sum of square penalty) at compile time rather than repeatedly computing them at runtime.

Problem Partition. The proximal algorithms in ∇-Prox rely on variable splitting that is based on a partition of two sets Ω and Ψ on every penalty function in a given problem[2], following the formulation in ProxImaL [2016]. In this formulation, the choice of splitting determines the introduction of auxiliary variables and can drastically affect the solver formulation and its performance.

$$g(\mathbf{x}) = \sum_{f_i \in \Omega} f_i(\mathbf{x}), \qquad h(\mathbf{z}) = \sum_{f_i \in \Psi} f_i(\mathbf{z}),$$

The optimal partition depends on the choice of specific algorithm and ∇-Prox would automatically detect and select the most proper one. For ADMM, a rule of thumb is that Ω often only contains quadratic terms so that the $\mathbf{x}$ subproblem can be reduced to a least squares problem.

Problem Scaling. The compiler includes an optional stage that performs the preconditioning on the split problem for accelerating the algorithm convergence and performance. By default, this is achieved by solving an equivalent problem whose variables are scaled by the spectral norm of the composited linear operator [Heide et al. 2016]. With the differentiable solver, our compiler also supports the *learning-based preconditioner* tailored for the given dataset, which usually exhibits better performance.

Code Generation. Once the problem is transformed, the next step is generating the solver algorithm code. All the algorithms in ∇-Prox exploit variable-splitting techniques and they share a common variable-splitting strategy to decouple every proxable function of the optimization objective into a set of separable easier-to-solve subproblems.

C.2 Extensible Sum Square

Typical proximal algorithms based on variable splitting all have one step that requires solving a linear system $\mathbf{Kx} = \mathbf{b}$. For example, ADMM solves generalized Problem (4) with multiple priors via alternatively solving three subproblems. The priors are properly split into two sets Ω and Ψ, where Ω only contains quadratic terms so that $\mathbf{x}$ subproblem can be reduced to a least squares problem. Then, it can be solved in closed form or via iterative methods such as conjugate gradient, based on whether linear operators of all quadratic terms have diagonal Gram matrices. In our framework, we consider another case where some quadratic terms can be solved in the closed form via special proximal operators even though the Gram matrices of its linear operator are not diagonal,

[2]Note the constraints c_j are converted into penalties f_j using indicator functions.

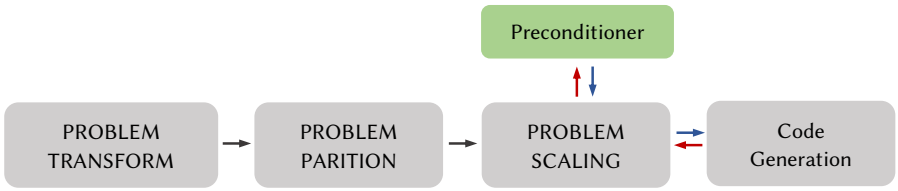


Fig. I. Overview of the stages of the general solving pipeline. It consists of two non-differentiable compilation stages for transforming the problem represented as DAG, and two differentiable stages for scaling and solving the problem where the gradient can backpropagate to the preconditioner or other learnable components.

which have been studied for image super-resolution [Chan et al. 2016], CSMRI [Wei et al. 2020], single photon imaging [Chan et al. 2016], and others.

To illustrate, we consider image super-resolution as an example where we use a single deep denoiser prior and a mean-square-error data-fidelity constraint as

$$\bar{\mathbf{x}} = \underset{\mathbf{x}}{\operatorname{argmin}} \ \frac{1}{2}||\mathbf{SHx} - \mathbf{y}||^2 + \lambda g(\mathbf{x}),$$

where $\mathbf{S}$ is a K-fold downsampling operator and $\mathbf{H}$ is a blurring operator, g denotes a deep denoiser prior and λ is corresponding penalty strength. When the ADMM is adopted, we solve this optimization problem by solving three subproblems as

$$\begin{aligned}
\mathbf{x}^{(k+1)} &= \underset{\mathbf{x}\in\mathbb{R}^n}{\operatorname{argmin}} \quad \tfrac{1}{2}||\mathbf{SHx} - \mathbf{y}||^2 + \tfrac{\rho}{2}\left\|\mathbf{x} - \tilde{\mathbf{x}}^{(k)}\right\|^2, \\
\mathbf{v}^{(k+1)} &= \underset{\mathbf{v}\in\mathbb{R}^n}{\operatorname{argmin}} \quad \lambda g(\mathbf{v}) + \tfrac{\rho}{2}\left\|\mathbf{v} - \widetilde{\mathbf{v}}^{(k)}\right\|^2, \\
\overline{\mathbf{u}}^{(k+1)} &= \overline{\mathbf{u}}^{(k)} + \left(\mathbf{x}^{(k+1)} - \mathbf{v}^{(k+1)}\right),
\end{aligned} \tag{20}$$

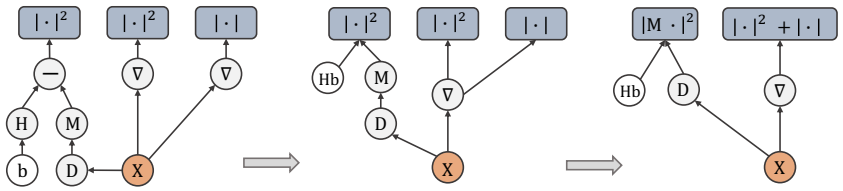
where $\overline{\mathbf{v}}^{(k)} \overset{\text{def}}{=} (1/\rho)\mathbf{v}^{(k)}$, $\widetilde{\boldsymbol{x}}^{(k)} \overset{\text{def}}{=} \boldsymbol{v}^{(k)} - \overline{\boldsymbol{u}}^{(k)}$ and $\widetilde{\boldsymbol{v}}^{(k)} \overset{\text{def}}{=} \boldsymbol{x}^{(k+1)} + \overline{\boldsymbol{u}}^{(k)}$ and the $\mathbf{x}$ update can be viewed as solving a linear system. In ProxImaL, the $\mathbf{x}$ update would be solved by iterative solvers as $\mathbf{SH}$ is not diagonal. In fact, the above $\mathbf{x}$ update could also be viewed as a proximal operator for the sum of square penalty function $\frac{1}{2}||\mathbf{SHx} - \mathbf{y}||^2$, and [Chan et al. 2016] show that it actually has a closed-form solution as

$$\operatorname{prox}_{\mathrm{f},\rho}(\tilde{\mathbf{x}}^{(k)}) = \rho^{-1}\mathbf{b} - \rho^{-1}\mathbf{G}^T\left(\mathcal{F}^{-1}\left\{\frac{\mathcal{F}(\mathbf{Gb})}{\left|\mathcal{F}\left(\tilde{h}_0\right)\right|^2 + \rho}\right\}\right) \tag{21}$$

where $\mathbf{G} = \mathbf{SH}, \mathbf{b} = \mathbf{G}^T\mathbf{y} + \rho\tilde{\mathbf{x}}^{(k)}$, $\tilde{h}_0$ is the 0^{th} polyphase component of the filter $\mathbf{HH}^T$.

A remaining issue of these special proximal operators is that they are mostly derived for optimization objectives with a single regularizer. To enable incorporating more regularizers, ∇-Prox introduces extensible sum squares that automatically extend existing closed-form sum square proximal operators that operate on a single regularizer for more regularizers. In a nutshell, ∇-Prox utilizes the fact that most special solutions (21) are derived from the least-square solution as

$$x^{(k+1)} = \left((\mathbf{HS})^T\mathbf{SH} + \rho I\right)^{-1}\left((\mathbf{HS})^T y + \rho\tilde{x}^{(k)}\right). \tag{22}$$



(a) Common subexpression Elimination
(b) Constant Folding
(c) Merge Proxable Function
(d) Absorb Linear Operator

Fig. II. ∇-Prox performs various optimization on the problem representation to eliminate unnecessary calculations to accelerate the forward and backward (gradient) computations.

When there are multiple regularizers and all their associated linear operators are identity, the least-square solution becomes

$$x^{(k+1)} = \left((\mathbf{HS})^T\mathbf{SH} + \rho\sum_j^N I\right)^{-1}\left((\mathbf{HS})^T y + \rho\sum_j^N \tilde{x}^{(k)}\right). \tag{23}$$

where N is the number of regularizers. We could see that solution (22) and (23) actually solve the same problem with different inputs (they only differ in I and $\tilde{x}^{(k)}$) and both of them can be replaced by fast solution (21). Since $\tilde{x}^{(k)}$ is the input of the proximal operator, the only thing we need to do is to change I in the proximal operator to reflect the number of regularizers.

The extensible sum square works exactly as discussed above. To implement it, ∇-Prox provides a base class that is supposed to be inherited by all the special proximal operators mentioned above. It automatically tracks the number of regularizers and emits the correct $\sum_j^N I$ as an instance property that can be used in the child class to implement their solving routines.

C.3 Eliminating Calculations

Proximal algorithms not only rely on the evaluation of proximal operators but also on the forward and adjoint operations of linear operators. To further increase efficiency, ∇-Prox encompasses all linear operators and proxable functions of a given problem into a DAG, which can be traversed to remove duplicate or unnecessary forward and gradient computations. Our implementation supports several optimizations, including constant folding, common subexpression elimination, proxable function fusion, and linear operator absorption.

For example, consider a combined penalty function for joint demosaicing and deconvolution with unaligned observation, data fidelity, and total variation regularizations,

$$r(\mathbf{x}) = \|\mathbf{Dx} - \mathbf{Hy}\|_2^2 + \|\nabla\mathbf{x}\|_2^2 + \|\nabla\mathbf{x}\|,$$

where ∇ denotes the gradient operator and $\mathbf{H}$ denotes the homography that aligns with the observation $\mathbf{y}$. As illustrated in Figure II, ∇-Prox is capable of intelligently performing a series of simplifications including a) fusing the common linear expression ∇x that can be evaluated only once and shared for two penalty functions; b) directly computing the aligned observation $\mathbf{Hy}$ at compile time instead of repeatedly evaluating them at runtime; c) identifying linear operators that can be absorbed into proxable functions, *e.g.*, $\mathbf{D}$ into the sum of squares, which might be helpful to circumvent iterative solving the least-square problem; d) merging proxable functions with the same linear operator *e.g.*, $\|\cdot\|_2^2$ and $\|\cdot\|$, into a new compound proxable function if it can be more efficiently evaluated. ∇-Prox would identify such a possibility and make the replacement if the resulting proxable function is more efficient than the previous ones.

D EXTENSIBILITY

In this section, we demonstrate the extensibility of ∇-Prox through step-by-step tutorials for creating custom linear operators, proxable functions, as well as proximal algorithms. Overall, ∇-Prox adopts the idea of object-oriented programming in Python, and all the operators and algorithms are represented as classes that inherit from a base class. For example, all the linear operators inherit from `LinOp` while all the proxable functions are a subclass of `ProxFn`. The common routines that can be shared are implemented in base classes by ∇-Prox, and the introduction of custom operators can be essentially achieved by implementing a few functions that are needed for evaluating the operators, *e.g.*, `forward` and `adjoint` in linear operator, and `prox` in proxable function. To maintain differentiability, all these operators and algorithms have to be correctly implemented to be differentiable as well. Fortunately, ∇-Prox integrates with PyTorch, so that users could utilize PyTorch's differentiable operators to build up custom operators for ∇-Prox.

D.1 Adding New Operators

Proxable Functions. The following code shows a template for defining a new proxable function. As previously mentioned, we define the function as a class inheriting from the base class `ProxFn`, and implement all the required methods. Then, ∇-Prox would handle all other things properly, so that the new proxable function can work with operators, algorithms, and training utilities of the existing system.

```
class new_func(ProxFn):
    def __init__(...):
        # Custom initialization code.

    def _prox(self, tau, v):
        # Code to compute the function's proximal operator.
        return ...

    def _eval(self, v):
        # (Optional) Code to evaluate the function.
        return ...

    def _grad(self, v):
        # (Optional) Code to compute the analytic gradient.
        return ...
```

Specifically, defining a new function only requires a method `_prox` to be implemented, which evaluates the proximal operator of the given function. Users can optionally implement the `_grad` function to provide a routine for computing the analytic gradient of the proxable function. This facilitates the algorithms that partially rely on the gradient evaluation, *e.g.*, proximal gradient descent [Bruck Jr 1975]. Besides, users could also implement the `_eval` method that computes the forwarding results of the proxable function if it is possible. ∇-Prox would take the `_eval` routine and compute the gradient with auto-diff if `_grad` is not implemented.

Linear Operators. Defining new linear operators mostly corresponds to defining the `forward` and `adjoint` routines. The following code shows the template for defining them. Similar to a proxable function, the operator is defined as a class inheriting from the base class `LinOp`.

```
class new_linop(LinOp):
    def __init__(...):
      # Custom initialization code.

    def forward(self, inputs):
        # Read from inputs, apply operator, and return outputs.
        return ...
```

```
    def adjoint(self, inputs):
        # Read from inputs, apply adjoint, and return outputs.
        return ...

    def is_diag(self, freq):
        # (Optional) Check if the linear operator is diagonalizable or
        # diagonalizable in the frequency domain.
        return ...

    def get_diag(self, x, freq):
        # (Optional) Return the diagonal/frequency diagonal matrix that
        # matches the shape of input x.
        return ...

    def params(self):
        # (Optional) Return the trainable parameters.
        return ...
```

By default, the linear operator is not diagonal. To introduce a diagonal linear operator, one must implement the `is_diag` and `get_diag` for checking the diagonalizability and acquiring the diagonal matrix. These methods facilitate ∇-Prox to construct more efficient solvers, *e.g.*, ADMM with closed-form matrix inverse for the least-square update.

Registering Learnable Parameters. By default, anything that inherits from `nn.Module` of PyTorch can be registered as trainable parameters of the linear operators or proxable functions. For other types of learnable components, *e.g.*, those hidden in the forward and adjoint function of black-box linear operators, users have to manually register them by either passing them into black-box operators or implementing the `params` method.

D.2 Adding New Proximal Algorithms

Extending ∇-Prox for more proximal algorithms is also easy and straightforward. We again define a new algorithm class that inherits from the base class `Algorithm`. For this particular case, the required methods that have to be implemented are `partition` and `_iter`, which stands for the problem partition and a single algorithm iteration. The `partition` takes a list of proxable functions and returns the splits of them as a list of `psi_fn` and `omega_fn` as previously discussed in Section C.1. For `_iter`, it is a single iteration of the proximal algorithm that takes an input of state and two parameters `rho` for penalty strength on multipliers and `lam` for proximal operators. The state is generally a list of variables including the auxiliary ones that an algorithm creates. ∇-Prox simply provides state as what returns in the previous execution of `_iter` or the initial state provided by `initialize` method.

```
class new_algorithm(Algorithm):
    def partition(cls, prox_fns: List[ProxFn]):
        # Perform problem partition on algorithm's need

    def __init__(...):
        # Custom initialization code.

    def _iter(self, state, rho, lam):
        # Code to compute the function's proximal operator.
        return ...

    def initialize(self, x0):
        # Return the initial state
        return ...

    def nparams(self):
        # (Optional) Return the number of hyperparameters of
```

```
        # this algorithm
        return ...

    def state_split(self):
        # (Optional) Return the split size of the packed state.
        # Useful for deep equilibrium/reinforcement learning.
        return ...
```

The implementations of `partition`, `initialize`, and `_iter` are generally enough for performing the evaluation of the proximal algorithm for a given problem. To integrate it with deep equilibrium learning (DEQ) and deep reinforcement learning (RL), users have to implement two additional helper methods, *i.e.*, `params` for counting the number of hyperparameters, and `state_split` for the structures of the state that returns by `_iter`. For example, assuming `_iter` returns the state as nested arrays like `[x,[v1,v2],[u1,u2]]`, the output of `state_split` should be `[1,[2],[2]]`. ∇-Prox exploits these properties to perform necessary packing and unpacking for the iteration states to achieve a unified interface for the internal DEQ and RL implementations.

D.3 Sanity Check

Dot Product Test for Linear Operator. Typically, it is not always easy to correctly implement the forward and adjoint operations of linear operators. To facilitate the testing of these operators, ∇-Prox provides an implementation of the dot-product test for verifying that the **`forward`** and **`adjoint`** are adjoint to each other. Basically, the idea of the dot-product test comes from the associative property of linear algebra, which gives the following equation,

$$y^T(Ax) = (A^T y)^T x$$

where x and y are randomly generated data, and A and A^T denote the forward and adjoint of the linear operator. ∇-Prox makes use of this property and generates a large number of random data to check if this equation always holds with respect to a given precision. To use this utility, users can call the `validate(linop, tol=1e-6)` and specify the tolerance of the difference between two sides of the equation.

E ADDITIONAL IMPLEMENTATION DETAILS

In this section, we present a comprehensive overview of a wide variety of built-in components implemented in ∇-Prox, which can be divided into i) built-in (linear/proximal) operators, ii) proximal algorithms, as well as iii) trainers for differentiable learned solvers.

E.1 Built-in Linear Operators

∇-Prox offers a set of commonly-used linear operators with all the `forward` and `adjoint` routines implemented in differentiable approaches. Here, we discuss the full set of linear operators that ∇-Prox currently supports.

Convolution. The `conv(x, psf)` operator calculates the circular convolution of a tensor x with n dimensions using a known kernel k with n dimensions. We implement the `conv` in the frequency domain. The adjoint is a circular convolution with the conjugate of the kernel. The `conv` is diagonal in the frequency domain.

Gradient. The `grad(x,dim)` operator calculates the discrete gradient of x along the specified dimension. The adjoint of this operator computes the negative divergence along the same dimension. The `grad` operator is diagonal in the frequency domain.

Subsample. The `subsample(x, steps)` operator selects elements from the input data using a step-based approach along each axis i, where the steps are defined as $\text{steps}_i - 1$ to steps_i. The adjoint of `subsample` reverses this process by inserting steps_i values before each entry along axis i. The `subsample` operator is Gram diagonal.

Element-wise multiplication. The `mul_elemwise(w, x)` operator performs element-wise multiplication between x and a fixed array w, and its adjoint also performs element-wise multiplication with w. The `mul_elemwise` operator is diagonal.

Scalar multiplication. The `scale(c,x)` operator denotes multiplication by a fixed scalar constant $c \in \mathbb{R}$, and its adjoint operation also corresponds to multiplication by c. The `scale` operator is diagonal.

Sum. The `sum(x1, x2, ..., xk)` operator performs the summation of k arrays $x_1, \ldots, x_k$, and its adjoint operation duplicates a single input into k different outputs.

Vstack. The `vstack(x1, x2, ..., xk)` operator stacks k arrays $x_1, \ldots, x_k$ vertically to form a single array, while its adjoint operation involves splitting a single array into k different subarrays. The `vstack` operator is diagonal.

Black box. The `black_box(forward, adjoint)` operator takes the function pointers of the forward and adjoint routines to build a block box linear operator. It can optionally accept a list of trainable parameters to facilitate the end-to-end training, though it is recommended to sub-classing the `LinOp` to build a new linear operator for complex cases.

E.2 Built-in Proximal Operators

∇-Prox offers a list of proximal operators for the commonly-used regularizers. Every proximal operator is offered as a class inherited from the base class `ProxFn`. Users could freely build their own custom proximal operator with minimum requirements. In the following, we provide an overview of the built-in ones.

Sum-squares. The `sum_squares(x)` computes the squared ℓ_2-norm, denoted as $f(x) = |x|_2^2$, for a vector x with n elements, where x belongs to the real number space $\mathbb{R}^n$. The proximal operator is given by

$$\mathbf{prox}_{\tau f}(\mathbf{v}) = \frac{1}{2\tau + 1}\mathbf{v}.$$

The function `weighted_sum_squares(D,x)` is a modified version that calculates the squared ℓ_2-norm, denoted as $f(x) = |Dx|_2^2$, for a vector x with n elements, where D is a diagonal matrix in the real number space $\mathbb{R}^{n\times n}$. The proximal operator is given by

$$\mathbf{prox}_{\tau f}(\mathbf{v}) = (2\tau D^T D + I)^{-1}\mathbf{v}.$$

The weighted variation is employed to absorb diagonal linear operators into the proximal function `sum_squares`.

ℓ_1-norm. The ℓ_1-norm is represented by the `norm1(x)` function, which computes $f(x) = |x|_1$, where $x \in \mathbb{R}^n$. The proximal operator for this function, denoted as $\mathbf{prox}_{\tau f}(\mathbf{v})$, is given by

$$\mathbf{prox}_{\tau f}(\mathbf{v})_i = \text{sign}(\mathbf{v}_i)\max\{|v_i| - \tau, 0\}, \quad i = 1, \ldots, n,$$

which is also known as soft-thresholding. There is a variant of the ℓ_1-norm, denoted as `weighted_norm1(D,x)`, which is defined as $f(x) = |Dx|_1$, where $D \in \mathbb{R}^{n\times n}$ is a diagonal matrix. The proximal operator for this weighted variant is given by the formula:

$$\mathbf{prox}_{\tau f}(\mathbf{v})_i = \text{sign}(\mathbf{v}_i)\max\{|v_i| - \tau|D_{ii}|, 0\}, \quad i = 1, \ldots, n.$$

The purpose of the weighted variant is to absorb diagonal linear operators into the `norm1` proxable function.

Poisson norm. The function `poisson_norm(x)` calculates the negative log-likelihood for a Poisson noise model, as described by

$$f(x) = \sum_{i=1}^{n} x_i - b_i \log(x_i) + \mathcal{I}_{(0,+\infty)}(x_i),$$

where b is a vector of non-negative real numbers with n components, x is a vector of real numbers with n components, and $\mathcal{I}_{(0,+\infty)}$ is the indicator function on the interval $(0, +\infty)$. The proximal operator is given by

$$\mathbf{prox}_{\tau f}(\mathbf{v})_i = \frac{\mathbf{v}_i - \tau}{2} + \sqrt{\tau b_i + (\tau - \mathbf{v}_i)^2/4}, \quad i = 1, \ldots, n.$$

The `weighted_poisson_norm(D,x)` function is a variant defined as

$$f(x) = \sum_{i=1}^{n} D_{ii}x_i - b_i \log(D_{ii}x_i) + \mathcal{I}_{(0,+\infty)}(D_{ii}x_i),$$

where $D \in \mathbb{R}^{n\times n}$ is a diagonal matrix. The proximal operator is given by

$$\mathbf{prox}_{\tau f}(\mathbf{v})_i = \frac{\mathbf{v}_i - \tau D_{ii}}{2} + \sqrt{\tau b_i + (\tau D_{ii} - \mathbf{v}_i)^2/4}, \quad i = 1, \ldots, n.$$

The weighted variation is employed to incorporate diagonal linear operators into the proximal function of the `poisson_norm` for better absorption.

Group ℓ_1-norm. The function `group_norm1(x, dims)` calculates the sum of ℓ_2-norms denoted as $f(x) = \sum_{i=1}^{p} |x_{g_i}|_2$, where $x \in \mathbb{R}^n$ and $g_1, \ldots, g_p$ is a partition of $1, \ldots, n$ obtained by flattening x along the specified dimensions. The proximal operator for this function is expressed as follows:

$$\mathbf{prox}_{\tau f}(\mathbf{v})_{g_i} = \mathbf{v}_{g_i}\max\{1 - \tau/\|\mathbf{v}_{g_i}\|_2, 0\}, \quad i = 1, \ldots, p,$$

which is also known as group soft-thresholding.

Nonnegativity constraint. The `nonneg(x)` function denotes the indicator $f(x) = \sum_{i=1}^{n} \mathcal{I}_{[0,+\infty)}(x_i)$, where $x \in \mathbb{R}^n$. The proximal operator is given by

$$\mathbf{prox}_{\tau f}(\mathbf{v})_i = \max\{\mathbf{v}_i, 0\}, \quad i = 1, \ldots, n.$$

The `weighted_nonneg(D,x)` function is a variant defined as

$$f(x) = \sum_{i=1}^{n} \mathcal{I}_{[0,+\infty)}(D_{ii}x_i),$$

where $D \in \mathbb{R}^{n \times n}$ is a diagonal matrix. The proximal operator is given by

$$\mathbf{prox}_{\tau f}(\mathbf{v})_i = \max\{D_{ii}\mathbf{v}_i, 0\}/D_{ii}, \quad i = 1, \ldots, n.$$

The weighted variant is used to absorb diagonal linear operators into the `nonneg` proxable function.

Denoising. The proximal operator, which incorporates a quadratic proximity term, can also be viewed as a Maximum a Posteriori (MAP) estimate for denoising Gaussian likelihoods, commonly referred to as Gaussian denoiser. For a Gaussian likelihood $p(\mathbf{v}|\mathbf{x})$ along with an arbitrary exponential prior $p(\mathbf{x})$.

$$p(\mathbf{v}|\mathbf{x}) \propto \exp\left(-\frac{\|\mathbf{x}-\mathbf{v}\|_2^2}{2\sigma^2}\right)$$
$$p(\mathbf{x}) \propto \exp\left(-\Gamma(\mathbf{x})\right),$$

By performing the MAP estimate on $p(\mathbf{x}|\mathbf{v})$, we obtain the proximal operator

$$\mathbf{prox}_{\sigma^2\Gamma}(\mathbf{v}) = \underset{\mathbf{x}}{\operatorname{argmin}} \left(\Gamma(\mathbf{x}) + \frac{1}{2\sigma^2}\|\mathbf{x}-\mathbf{v}\|_2^2\right).$$

This could be solved by any Gaussian denoising algorithms $\{D_\sigma^\Gamma : \sigma > 0\}$, which estimate $\mathbf{x}_0$ from $\mathbf{x}_0 + \sigma\mathbf{z}$ with $\mathbf{z} \sim \mathcal{N}(0, \mathbb{I})$, as

$$\mathbf{prox}_{\tau\Gamma}(\mathbf{v}) = D_{\sqrt{\tau}}^\Gamma(\mathbf{v}).$$

Please note that it is not obligatory to compute Γ in order to evaluate the proximal operator. We only need D_σ^Γ that performs Gaussian denoising to implicitly model the data prior, which enables great flexibility leveraging the latest techniques, *e.g.*, deep neural networks, as black-box proximal operators. The proximal operator for denoising is `deep_prior` in ∇-Prox.

E.3 Build-in Proximal Algorithms.

Half-quadratic Splitting. Half-Quadratic Splitting (HQS) is a simplified version of ADMM that removes the Lagrangian multiplier. Therefore, it only contains two updates without the dual update of the multiplier. The pseudo-code of it is given in Algorithm 2. Our compiler uses the default hyper-parameters $\rho = 1 \times 10^{-5}$ and $\sigma = 0.14$. The default partition strategy for HQS is the same as for ADMM.

Algorithm 2 Half-Quadratic Splitting to solve Problem (1) of the main paper

1: Initialization: $\rho^0 > 0$, $\rho_{\max} > 0$, $(\mathbf{x}^0, \mathbf{z}^0)$.
2: **for** $k = 1$ to V **do**
3: $\quad \mathbf{x}^{k+1} = \underset{\mathbf{x}}{\operatorname{argmin}} \ \sum_{i\in\Omega} f_i(\mathbf{K}_i\mathbf{x}) + \rho^k \sum_{j\in\Psi} \|\mathbf{K}_j\mathbf{x} - \mathbf{z}_j\|_2^2$
4: $\quad \mathbf{z}_j^{k+1} = \mathbf{prox}_{\frac{f_j}{\sigma^k}}(\mathbf{K}_j\mathbf{x}_j^{k+1}) \quad \forall j \in \Psi$
5: **end for**

Linearized ADMM. . The linearized ADMM is a linearized version of ADMM whose pseudo-code is given in Algorithm 3. Our compiler uses the default hyper-parameters as it is for HQS and ADMM. The default partition strategy for linearized ADMM is the same as for ADMM.

Algorithm 3 Linearized ADMM to solve Problem (1) of the main paper

Initialization: $\mu > \rho\|\mathbf{K}\|_2^2$, $(\mathbf{x}^0, \mathbf{z}^0, \lambda^0)$.
for $k = 1$ to V **do**
  $\mathbf{x}^{k+1} = \mathbf{prox}_{\frac{g}{\mu}}(\mathbf{x}^k - (\rho/\mu)K^T(K\mathbf{x}^k - \mathbf{z}^k + \lambda^k))$
  $\mathbf{z}_j^{k+1} = \mathbf{prox}_{\frac{f_j}{\sigma}}(\mathbf{K}_j\mathbf{x}_j^{k+1} + \lambda_j^k) \quad \forall j \in \Psi$
  $\lambda_j^{k+1} = \lambda_j^k + (\mathbf{K}_j\mathbf{x}^{k+1} - \mathbf{z}_j^{k+1}) \quad \forall j \in \Psi$
end for

Pock-Chambolle. The Pock-Chambolle, as shown in Algorithm (4), is in fact the linearized ADMM applied to the dual of Problem (1) [Chambolle and Pock 2011]. The dual problem involves the conjugates of $f_1, \ldots, f_I$ from Problem (1). The proximal operator for the conjugate f_i^* can be evaluated using the proximal operator for f_i and Moreau's Identity [Moreau 1965].

Algorithm 4 Pock-Chambolle to solve Problem (1) of the main paper

Initialization: $\sigma\tau\|\mathbf{K}\|_2^2 < 1$, $\theta \in [0, 1]$, $(\mathbf{x}^0, \mathbf{z}^0)$, $\bar{\mathbf{x}}^0 = \mathbf{x}^0$.
for $k = 1$ to V **do**
  $\mathbf{z}_j^{k+1/2} = \mathbf{z}_j^k + \sigma\mathbf{K}_j\bar{\mathbf{x}}^k \quad \forall j \in \Psi$
  $\mathbf{z}_j^{k+1} = \mathbf{z}_j^{k+1/2} - \sigma\mathbf{prox}_{f_j/\sigma}\left(\mathbf{z}_j^{k+1/2}/\sigma\right) \quad \forall j \in \Psi$
  if $\Omega = \{f_i\}$ **then**
    $\mathbf{x}^{k+1} = \mathbf{prox}_{\tau f_i}\left(\mathbf{x}^k - \tau\mathbf{K}^T\mathbf{z}^{k+1}\right)$
  else
    $\mathbf{x}^{k+1} = \mathbf{x}^k - \tau\mathbf{K}^T\mathbf{z}^{k+1}$
  end if
  $\bar{\mathbf{x}}^{k+1} = \mathbf{x}^{k+1} + \theta\left(\mathbf{x}^{k+1} - \mathbf{x}^k\right)$
end for

Proximal Gradient Descent. Proximal Gradient Descent decomposes the optimization of the given objective into two parts where the gradient descent is used for the convex and differentiable part and the proximal mapping is used for the potentially non-differentiable part. The pseudo-code of it is given in Algorithm 5. Proximal Gradient Descent is only applicable for two regularizers.

Algorithm 5 Proximal Gradient Descent to solve Problem (1) of the main paper

Initialization: ρ, σ, $(\mathbf{x}^0, \mathbf{z}^0)$.
for $k = 1$ to V **do**
  $\mathbf{x}^{k+1} = \mathbf{z}^k - \rho\nabla g(\mathbf{z}^k)$
  $\mathbf{z}_j^{k+1} = \mathbf{prox}_{\frac{f_j}{\sigma^k}}(\mathbf{x}_j^{k+1})$
end for

E.4 Training Interface for Learned Solvers

Thanks to the transparent design of ∇-Prox, the training of the learned solver can be achieved in a unified training interface, *i.e.*, `train(args, dataset, solver, step_fn)`. The solver passed into `train` can be either an unrolled solver, DEQ solver, or RL solver.

```
solver = specialize(solver, 'unroll', share=False)
# or we could directly wrap the solver.
solver = UnrolledSolver(solver, share=False)
solver = DEQSolver(solver)
solver = RLSolver(solver)
# setup the forward step.
def step_fn(batch):
    target, b = batch
    y.value = b
    pred = solver.solve(x0=b, rhos=rhos, lams={reg_term: sigmas}, max_iter=max_iter)
   return target, pred
# call the utility to train it.
train(args, dataset, solver, step_fn)
```

Specifically, training a learned solver is straightforward in ∇-Prox. The first step for it is transforming the solver by calling `specialize` or wrapping the existing solver by a solver wrapper, *e.g.*, `DEQSolver`. The resulting solver exhibits the same interface as the original one so that the training can be done with a conventional training pipeline of learning-based approaches. In ∇-Prox, we provide a utility `train_deq` to facilitate fast prototyping. Basically, we pass in the arguments specifying the training options, training dataset, solver with learnable parameters, and a step function that unwraps the data and evaluates the forward model. The intermediate results are periodically saved for inspecting the training process. Our training utility is general and easy to be modified, which we believe can be a good starting point for primary testing and training for different base solvers. Internally, `train` calls the other training utilities, `train_unroll`, `train_deq`, and `train_rl` based on the type of learned solver. Most of part of these utilities are the same except for some special method-specific treatments, *e.g.*, building the critic network for RL training, and checking the over-flow loss for DEQ.

F LEARN ∇-PROX WITH EXAMPLES

In this section, we provide step-by-step examples for compiling the Problem (24) discussed in the main paper.

$$\mathbf{x}^* = \underset{\mathbf{x}}{\operatorname{argmin}} \ \|\mathbf{D}(\mathbf{x};\, \theta_{DOE}) - \mathbf{y}\|_2^2 + r(\mathbf{x}; \theta_r) \tag{24}$$

$$\begin{aligned} f_1(\mathbf{v}) &= \|\mathbf{v} - \mathbf{y}\|_2^2, \qquad & \mathbf{K}_1 &= \mathbf{D}(\cdot;\, \theta_{DOE}) \\ f_2(\mathbf{v}) &= \lambda g(\mathbf{v}; \theta_r), & \mathbf{K}_2 &= \mathbf{I} \\ f_3(\mathbf{v}) &= I_{[0,\infty)}(\mathbf{v}), & \mathbf{K}_3 &= \mathbf{I} \end{aligned} \tag{25}$$

With only a few lines of code, ∇-Prox can compile the problem into various differentiable solvers that can be used independently or be integrated into an existing differentiable pipeline, *e.g.*, optimization together with a post-processing network.

We first consider joint deconvolution and denoising as a simplified version of the Problem (24), that is with a *fixed* PSF instead of a learned one. The forward model of this problem can be defined in ∇-Prox with `conv(x, psf)`. Given an observation `y`, our goal is to find the unknown image $\hat{\mathbf{x}}$ by minimizing an objective function, which can concisely be described in ∇-Prox with almost the same syntax as the mathematical objective.

```
          x = Variable()
  data_term = sum_squares( conv(x, psf) - y )
 prior_term = deep_prior(x, unet)
  objective = data_term + prior_term + nonneg(x)
```

∇-Prox facilitates experimenting with regularizations. For example, users can trivially change the plug-and-play prior with the denoiser FFDNet [Zhang et al. 2018] as `deep_prior(x, 'ffdnet')`, or use a total variation prior with `norm1(grad(x))`. Solving the objective is achieved by wrapping it into a `Problem` class and calling the `solve` method by specifying the desired algorithms, *e.g.*, ADMM in this case.

```
        p = Problem( objective )
      out = p.solve(method='admm')
```

Users are allowed to pass more options to configure this process, *e.g.*, the initial guess, max iterations, and algorithm parameters, with keyword arguments of `p.solve(x0=..., rhos=..., max_iters=...)`.

We note that `p.solve` is a differentiable routine that can be integrated with PyTorch [Paszke et al. 2019] for downstream learning-based tasks. Despite that, it is usually more efficient to use the compiled solver with our `Placeholder` and `compile` functionalities. To illustrate this, we return back to Problem (24) with learnable PSF, that is

```
          y = Placeholder(input)
  data_term = sum_squares( conv(x, psf_DOE) - y )
 prior_term = deep_prior(x, unet)
  objective = data_term + prior_term + nonneg(x)
          s = compile(objective, method='admm')
```

To learn this PSF along the rest of the parameters, one immediate choice to make the solver differentiable may be unrolling which can be immediately achieved by fixing the `max_iters` argument of `p.solve`. To provide finer-grained control, ∇-Prox provides a primitive `specialize` to specialize the existing solver into an unrolled one with independent parameters for each iteration.

```
s2 = specialize(s, method='unroll', shared=False)
```

The resulting unrolled solver can then allow for fine-tuning the learning-based PnP prior or even training a learnable linear operator, *e.g.*, the above learnable PSF with a physical imaging model `psf_DOE`.

As discussed in the previous paragraphs, unrolling for many iterations becomes intractable. To this end, ∇-Prox supports incorporating deep equilibrium learning for many iterations which, from a user perspective, is as immediate as algorithm unrolling using the **specialize** primitive.

```
s3 = specialize(s, method='deq')
```

The DEQ solver is memory-effective and can be trained with the utility `train_deq(s3, dataset, **cfg)` provided by ∇-Prox, via specifying the solver, training dataset, and configurations. Being able to compile differentiable solvers, ∇-Prox is also capable of automatically estimating the parameters and the terminal time. Specifically, we extend the **specialize** primitive to easily transform the solver to the tuning-free (TF) one that can be learned with reinforcement learning [Wei et al. 2020].

```
s4 = specialize(s, method='rl', policy='resnet')
```

Training these TF solvers can also be achieved with the built-in training utility `train_rl(s4, dataset, **cfg)`.

All the solvers in ∇-Prox, including the vanilla one and the specialized ones, contain a `s.solve` method that shares the same function signature. Users can then seamlessly switch between different specializations, which provides flexibility for rapid prototyping for diverse applications.

G ADDITIONAL DETAILS FOR INTEGRATED ENERGY SYSTEM PLANNING

G.1 General Modeling

The modeling and optimization framework for integrated energy system transformation pathways is a multi-period capacity expansion and system operation problem. Conceptually, the framework follows a generic structure and logic to account for the necessity of modeling the vital integration of energy and non-energy commodities and their underlying sectors, markets, infrastructures, and technologies. A concise description and formulation with sets and indices, constraints, and the objective function are given below.

G.2 Sets and Indices

The model formulation requires multiple sets covering the system representation domains of integrated energy systems and the multi-fold interaction between components.

$p \in P$	Planning periods,
$t \in T$	Time steps (discrete and equidistant),
$n \in N$	Nodes,
$c \in C$	Commodities,
$u \in U$	Converters,
$s \in S$	Storage systems,
$w \in W$	Sources,
$d \in D$	Sinks,
U_c	Subset of converters U interacting with commodity c,
S_c	Subset of storage systems S interacting with commodity c,
$l \in L_c$	Links for commodity c (two-dimensional),
$l \in \overline{L}_c$	$\overline{L}_c := \{ (n,m) \in L_c : n > m \}$,
$\delta_c^{\text{out}}(n)$	$\delta_c^{\text{out}}(n) := \{ l \in L_c : \exists m \text{ with } l = (n,m) \}$,
$\delta_c^{\text{in}}(n)$	$\delta_c^{\text{in}}(n) := \{ l \in L_c : \exists m \text{ with } l = (m,n) \}$.

The planning periods P represent stages of the transformation path under investigation (*e.g.*, the years 2025 to 2055 in 10-year steps). Time steps T typically capture a full meteorological year in hourly resolution to capture all seasons and reflect weather-dominated renewable generation technologies as well as end-use demand patterns. Nodes N may represent network nodes, price zones, jurisdictions, or regions at the global scale. Commodities C include energy and non-energy commodities. Energy commodities include electricity, hydrogen, methane, ammonia, liquids, transport demands, as well as heating and cooling for residential and non-residential applications. Non-energy commodities are carbon-dioxide, whose circular management becomes very relevant in climate-neutral settings. Converters U serve comprise several technologies of integrated energy systems. They include thermal generator (*e.g.*, open- and closed-cycle gas turbines, nuclear) and (variable) renewable generator (*e.g.*, solar photovoltaic, on- and offshore wind, biomass) technologies. Moreover, converter units describe several commodity conversion systems (*e.g.*, heat pump, boiler, electric vehicle, electrolyzer, methanation, and carbon capture units). Storage systems S describe commodity-specific storage technologies (*e.g.*, battery, pumped-hydro, gas storage for hydrogen or gas, thermal storage, or carbon storage). Sources W allow the energy system planning problem to explicitly represent potential sources of commodities to supply the individual demands (*e.g.*, import of fossil or renewable hydrogen or liquids at given costs). Sinks D introduce inelastic or flexible end-use demands for various commodities to the system planning problem (*e.g.*, conventional electricity demand, residential and non-residential heating and cooling demands, feedstock demands for hydrogen, methane or carbon dioxide). Links L allow the transport of commodities via energy and non-energy networks between nodes (*e.g.*, electricity, hydrogen, methane, carbon dioxide).

G.3 Decision Variables

The decision variables describe capacity expansion and dispatch decisions. The model makes endogenous capacity investment decisions for the converter, storage system, and link components for every considered planning period. The dispatch variables of all relevant components reflect the scheduling decisions for operating and clearing the integrated system for every considered planning period and time step.

$\overline{x}_{u,n,p} \in \mathbb{R}^+$	$\forall u \in U, \forall n \in N, \forall p \in P$	Converter capacity,
$\overline{x}^+_{u,n,p} \in \mathbb{R}^+$	$\forall u \in U, \forall n \in N, \forall p \in P$	Converter capacity expansion,
$x_{u,n,p,t} \in \mathbb{R}^+$	$\forall u \in U, \forall n \in N, \forall p \in P, \forall t \in T$	Converter dispatch,
$\overline{x}_{s,n,p} \in \mathbb{R}^+$	$\forall s \in S, \forall n \in N, \forall p \in P$	Storage capacity,
$\overline{x}^+_{s,n,p} \in \mathbb{R}^+$	$\forall s \in S, \forall n \in N, \forall p \in P$	Storage capacity expansion,
$x^{\text{in}}_{s,n,p,t} \in \mathbb{R}^+$	$\forall s \in S, \forall n \in N, \forall p \in P, \forall t \in T$	Storage injection dispatch,
$x^{\text{out}}_{s,n,p,t} \in \mathbb{R}^+$	$\forall s \in S, \forall n \in N, \forall p \in P, \forall t \in T$	Storage withdrawal dispatch,
$x^{\text{lvl}}_{s,n,p,t} \in \mathbb{R}^+$	$\forall s \in S, \forall n \in N, \forall p \in P, \forall t \in T$	Storage level dispatch,
$\overline{x}_{l,c,p} \in \mathbb{R}^+$	$\forall l \in \overline{L}_c, \forall c \in C, \forall p \in P$	Link capacity,
$\overline{x}^+_{l,c,p} \in \mathbb{R}^+$	$\forall l \in \overline{L}_c, \forall c \in C, \forall p \in P$	Link capacity expansion,
$x_{l,c,p,t} \in \mathbb{R}^+$	$\forall l \in L_c, \forall c \in C, \forall p \in P, \forall t \in T$	Link dispatch,
$x_{w,c,n,p,t} \in \mathbb{R}^+$	$\forall w \in W, \forall c \in C, \forall n \in N, \forall p \in P, \forall t \in T$	Source dispatch,
$x_{d,c,n,p,t} \in \mathbb{R}^+$	$\forall d \in D, \forall c \in C, \forall n \in N, \forall p \in P, \forall t \in T$	Sink dispatch.

G.4 Constraints

The modeling framework requires a set of linear constraints to incorporate characteristics and limits for capacity expansion and operational dispatch decisions of its model components. Equations (26) to (28) define lower and upper limits on the capacity expansion decisions for converter, storage, and link components from geographically and technically feasible potentials. Equations (29) to (34) impose the corresponding capacity expansion continuities between individual planning periods throughout the considered transformation pathway. Equations (35) to (42) enforce the individual dispatch characteristics and technology availability limits including the storage level

continuity and commodity transport via the corresponding energy and non-energy networks between nodes.

$$\underline{X}_{u,n,p} \leq \overline{x}_{u,n,p} \leq \overline{X}_{u,n,p} \qquad \forall u \in U, \forall n \in N, \forall p \in P \tag{26}$$

$$\underline{X}_{s,n,p} \leq \overline{x}_{s,n,p} \leq \overline{X}_{s,n,p} \qquad \forall s \in S, \forall n \in N, \forall p \in P \tag{27}$$

$$\underline{X}_{l,c,p} \leq \overline{x}_{l,c,p} \leq \overline{X}_{l,c,p} \qquad \forall l \in \overline{L}_c, \forall c \in C, \forall n \in N, \forall p \in P \tag{28}$$

$$\overline{x}_{u,n,p} = \overline{x}_{u,n,p-1} + \overline{x}^{+}_{u,n,p} \qquad \forall u \in U, \forall n \in N, \forall p \in P \setminus \{p_0\} \tag{29}$$

$$\overline{x}_{s,n,p} = \overline{x}_{s,n,p-1} + \overline{x}^{+}_{s,n,p} \qquad \forall s \in S, \forall n \in N, \forall p \in P \setminus \{p_0\} \tag{30}$$

$$\overline{x}_{l,c,p} = \overline{x}_{l,c,p-1} + \overline{x}^{+}_{l,c,p} \qquad \forall l \in \overline{L}_c, \forall c \in C, \forall p \in P \setminus \{p_0\} \tag{31}$$

$$\overline{x}_{u,n,p_0} = \overline{X}_{u,n,0} + \overline{x}^{+}_{u,n,p_0} \qquad \forall u \in U, \forall n \in N \tag{32}$$

$$\overline{x}_{s,n,p_0} = \overline{X}_{s,n,0} + \overline{x}^{+}_{s,n,p_0} \qquad \forall s \in S, \forall n \in N \tag{33}$$

$$\overline{x}_{l,c,p_0} = \overline{X}_{l,c,0} + \overline{x}^{+}_{l,c,p_0} \qquad \forall l \in \overline{L}_c, \forall c \in C \tag{34}$$

$$\underline{A}_{u,n,p,t}\, \overline{x}_{u,n,p} \leq x_{u,n,p,t} \leq \overline{A}_{u,n,p,t}\, \overline{x}_{u,n,p} \qquad \forall u \in U, \forall n \in N, \forall p \in P, \forall t \in T \tag{35}$$

$$\underline{A}_{s,n,p,t}\, \overline{x}_{s,n,p} \leq x^{\text{in}}_{s,n,p,t} \leq \overline{A}_{s,n,p,t}\, \overline{x}_{s,n,p} \qquad \forall s \in S, \forall n \in N, \forall p \in P, \forall t \in T \tag{36}$$

$$\underline{A}_{s,n,p,t}\, \overline{x}_{s,n,p} \leq x^{\text{out}}_{s,n,p,t} \leq \overline{A}_{s,n,p,t}\, \overline{x}_{s,n,p} \qquad \forall s \in S, \forall n \in N, \forall p \in P, \forall t \in T \tag{37}$$

$$\underline{A}_{s,n,p,t}\, \overline{x}_{s,n,p} \leq \Phi_s\, x^{\text{lvl}}_{s,n,p,t} \leq \overline{A}_{s,n,p,t}\, \overline{x}_{s,n,p} \qquad \forall s \in S, \forall n \in N, \forall p \in P, \forall t \in T \tag{38}$$

$$x^{\text{lvl}}_{s,n,p,t+1} = (1 - \Lambda_s)\, x^{\text{lvl}}_{s,n,p,t} + \left(\eta^{\text{in}}_s x^{\text{in}}_{s,n,p,t} - \frac{x^{\text{out}}_{s,n,p,t}}{\eta^{\text{out}}_s} \right) \Delta^{\text{t}} \qquad \forall s \in S, \forall n \in N, \forall p \in P, \forall t \in T \tag{39}$$

$$0 \leq x_{(n,m),c,p,t} + x_{(m,n),c,p,t} \leq \overline{A}_{l,c,p,t}\, \overline{x}_{l,c,p} \qquad \forall (n,m) \in \overline{L}_c, \forall c \in C, \forall p \in P, \forall t \in T \tag{40}$$

$$\underline{D}_{d,c,n,p,t} \leq x_{d,c,n,p,t} \leq \overline{D}_{d,c,n,p,t} \qquad \forall d \in D, \forall c \in C, \forall n \in N, \forall p \in P, \forall t \in T \tag{41}$$

$$\underline{W}_{w,c,n,p,t} \leq x_{w,c,n,p,t} \leq \overline{W}_{w,c,n,p,t} \qquad \forall w \in W, \forall c \in C, \forall n \in N, \forall p \in P, \forall t \in T \tag{42}$$

Here, $\underline{X}_{(\cdot),(\cdot),p} \in \mathbb{R}^+$ and $\overline{X}_{(\cdot),(\cdot),p} \in \mathbb{R}^+$ are lower and upper capacity potentials, respectively, and $\overline{X}_{(\cdot),n,0} \in \mathbb{R}^+$ is the already existing capacity in initial planning period p_0 for component $(\cdot)$. Moreover, $\underline{A}_{(\cdot),(\cdot),p,t} \in [0,1]$ and $\overline{A}_{(\cdot),(\cdot),p,t} \in [0,1]$ are location-, planning period-, and time step-dependent parameters for minimum and maximum technology availability. For storage components, $\Phi_s \in \mathbb{R}^+$ denotes storage power-to-energy ratio, $\Lambda_s \in [0,1]$ are scalar parameters for specific self-discharge losses, and $\eta^{\text{in/out}}_s \in [0,1]$ are scalar parameters for storage injection and withdrawal efficiencies, and Δ^{t} is the constant duration of the considered time step, e.g., 1 hour. For sinks and sources, $\underline{D}_{d,c,n,p,t} \in \mathbb{R}^+$ and $\overline{D}_{d,c,n,p,t} \in \mathbb{R}^+$ as well as $\underline{W}_{w,c,n,p,t} \in \mathbb{R}^+$ and $\overline{W}_{w,c,n,p,t} \in \mathbb{R}^+$ are location-, planning period-, and time step-dependent parameters for minimum and maximum sink demands and source supply. Note that, for simplicity, upper or lower budget constraints are omitted here but can easily be integrated via constraints summing over one or more dimensions. For instance, global annual emission budgets could be achieved by summing over all time steps $t \in T$ of a planning period and all nodal sinks d for such a commodity c.

The nodal commodity balance or clearing constraint in Equation (43) ensures that the supply and demand including relevant storage components and in- and outgoing links to other nodes are in equilibrium for every

planning period and time step.

$$0 = \sum_{u \in U_c} \left(\Theta^{\text{op}}_{u,c} - \Theta^{\text{ip}}_{u,c}\right) x_{u,n,p,t} + \sum_{s \in S_c} \left(x^{\text{out}}_{s,n,p,t} - x^{\text{in}}_{s,n,p,t}\right) - \sum_{d \in D} x_{d,c,n,p,t} + \sum_{w \in W} x_{w,c,n,p,t} + \sum_{l \in \delta^{\text{in}}_c(n)} x_{l,c,p,t} - \sum_{l \in \delta^{\text{out}}_c(n)} \frac{1}{\eta_l} x_{l,c,p,t} \qquad \forall n \in N, \forall c \in C, \forall p \in P, \forall t \in T \quad (43)$$

Here, $\Theta^{\text{ip}}_{u,c} \in \mathbb{R}$ and $\Theta^{\text{op}}_{u,c} \in \mathbb{R}$ are commodity in- and output factors of converter component u, and $\eta_l \in [0, 1]$ is a scalar parameter for line-length-specific (linear) transmission loss factors for the commodity transport via link l.

G.5 Objective function and optimization problem

The objective of the modeling framework is to minimize the total system cost of the energy system transformation pathway, incorporating investment costs of multi-period capacity expansion planning and operational costs of supplying end-use demands given the constraints outlined above. The period-specific investment cost function $f^{\text{ic}}_p(x)$ gathers all investment costs for expanding the capacities of converters, storage systems, and links that allow for commodity transport between the considered nodes, which writes as

$$f^{\text{ic}}_p(x) = \sum_{n \in N} \left(\sum_{u \in U} \pi^{\text{ic}}_{u,p} \, x^{+}_{u,n,p} + \sum_{s \in S} \pi^{\text{ic}}_{s,p} \, x^{+}_{s,n,p} \right) + \sum_{c \in C} \sum_{l \in \overline{L}_c} \pi^{\text{ic}}_{l,p} \, x^{+}_{l,c,p} \qquad \forall p \in P, \quad (44)$$

where $\pi^{\text{ic}}_{(\cdot),p} \in \mathbb{R}$ are scalar parameters for (annualized) investment cost. The period-specific operational cost function $f^{\text{oc}}_{p,t}(x)$ describes the cost of dispatching all relevant system components and is defined for every time step by

$$f^{\text{oc}}_{p,t}(x) = \sum_{n \in N} \left(\sum_{u \in U} \pi^{\text{oc}}_{u,p,t} \, x_{u,n,p,t} + \sum_{s \in S} \pi^{\text{oc}}_{s,p,t} \left(x^{\text{in}}_{s,n,p,t} + x^{\text{out}}_{s,n,p,t} \right) + \sum_{c \in C} \left(\sum_{d \in D} \pi^{\text{oc}}_{d,c,n,p,t} \, x_{d,c,n,p,t} + \sum_{w \in W} \pi^{\text{oc}}_{w,c,n,p,t} \, x_{w,c,n,p,t} \right) \right) \qquad \forall t \in T, \forall p \in P, \quad (45)$$

where $\pi^{\text{oc}}_{(\cdot),p,t} \in \mathbb{R}$ and $\pi^{\text{oc}}_{(\cdot),c,n,p,t} \in \mathbb{R}$ are scalar parameters for component-, planning period-, and partly time step-dependent (marginal) operational costs. With the objective function components defined in Equations (45) and (44), the resulting linear programming problem writes as

$$\begin{aligned} \min f(x) = \quad & \sum_{p \in P} \xi_p \left(f^{\text{ic}}_p(x) + \sum_{t \in T} f^{\text{oc}}_{p,t}(x) \right) \\ \text{s.t.} \quad & \text{Equations (26) to (28) (\textit{Capacity potential restrictions})} \\ & \text{Equations (29) to (34) (\textit{Expansion continuities})} \\ & \text{Equations (35) to (42) (\textit{Dispatch characteristics and limits})} \\ & \text{Equation (43) (\textit{Nodal commodity clearing})} \end{aligned} \quad (46)$$

where $\xi_p \in \mathbb{R}$ is a period-specific discount factor to account for the time value of money and perpetuity in the multi-period planning problem.

G.6 Convergence Loss

After the formal introduction of the integrated energy system planning problem, we now present the convergence loss function mentioned in Section 5.4 of the main paper. Consider the following linear programming problem,

$$\begin{aligned} \min \quad & c^T x \\ \text{subject to} \quad & Ax \in C \end{aligned} \tag{47}$$

where $x, c \in \mathbb{R}^n$ denote the decision variable and the cost vector respectively, $A \in^{m\times n}$ is the constraint matrix and the set C takes the form

$$C = [l, u] = \{z \in \mathbb{R}^m \mid l_i \leq z_i \leq u_i,\ i = 1, ..., m\}$$

It can be verified that the complex integrated energy system model (46) with its polyhedral constraint set can reduce to the simple form of (47). To leverage proximal algorithms to tackle this problem, we rewrite the problem (47) by introducing an additional decision variable $z \in \mathbb{R}^m$ to obtain the equivalent problem

$$\begin{aligned} \min \quad & c^T x \\ \text{subject to} \quad & Ax = z \\ & z \in C. \end{aligned} \tag{48}$$

The optimality conditions of problem (48) are be written as

$$Ax = z, \tag{49}$$

$$c + A^T y = 0, \tag{50}$$

$$z \in C,\ y = N_C(z), \tag{51}$$

where $y \in \mathbb{R}^m$ is the Lagrange multiplier with respect to the constraint $Ax = z$ and N_C denotes the normal cone of C at z. If there exists x, z, y that satisfy the conditions above, then we say that (x, z) is a *primal* and y is a *dual* solution to problem (48). We define the primal and dual residuals of problem (47) as

$$r_{\text{prim}} = Ax - z, \tag{52}$$

$$r_{\text{dual}} = c + A^T y. \tag{53}$$

which are essential indicators related to a bound on the objective suboptimality [Boyd et al. 2011]. In proximal algorithms such as ADMM, the termination criterion can be reasonably derived based on the the primal and dual residuals r_{prim} and r_{dual}, *i.e.*,

$$\|r_{\text{prim}}\| \leq \epsilon^{\text{prim}} \text{ and } \|r_{\text{dual}}\| \leq \epsilon^{\text{dual}} \tag{54}$$

where $\|\cdot\|$ is a norm operator that can be taken in various forms (1, 2, ∞-norm), ϵ^{prim} and ϵ^{dual} are feasibility tolerances for the primal and dual feasibility conditions, respectively. In practice, these tolerances are selected as a combination of an absolute and relative criterion, such as

$$\epsilon^{\text{prim}} = \epsilon^{\text{abs}} + \epsilon^{\text{rel}} \max\{\|Ax^k\|, \|z^k\|\} \tag{55}$$

$$\epsilon^{\text{dual}} = \epsilon^{\text{abs}} + \epsilon^{\text{rel}} \max\{\|A^T y^k\|, \|c\|\}, \tag{56}$$

where $\epsilon^{\text{abs}} > 0$ is an absolute tolerance and $\epsilon^{\text{rel}} > 0$ is a relative tolerance, In the above formulation, one can immediately realize that to speed up the practical convergence, the proximal solver should meet the stopping

criterion (54) as soon as possible. Building upon this observation, we propose a convergence loss function that encourages solvers to actively chase the stopping criterion, *i.e.*,

$$\mathcal{L}_{\text{converge}} = \log\left(\frac{\|r_{\text{prim}}\|}{\epsilon^{\text{prim}}}\right) + \log\left(\frac{\|r_{\text{dual}}\|}{\epsilon^{\text{dual}}}\right) \tag{57}$$

Minimizing this loss function naturally meets the stopping criterion (54). Since the operands involved in (57) are computed from optimization variables x, z, y, the gradient can be directly backpropagated downstream to algorithm parameters for fast adaptation thanks to the differentiability design of ∇-Prox.

H ADDITIONAL APPLICATION SETUPS AND RESULTS

In this section, we provide more details about the experimental setup for the applications presented in the main paper. Additional quantitative and visual results are also provided.

H.1 End-to-End Computational Optics

Here, we provide the training details of the end-to-end computational single thin-lens imaging about ∇-Prox and the competing methods. Specifically, all the models are trained for 100 epochs with the AdamW optimizer at a constant learning rate 1×10^{-4} for the first 50 epochs and 1×10^{-5} for the last 50 epochs. The training can generally be finished in a few hours depending on the used hardware. ∇-Prox leverages a hybrid model- and learning-based PnP solver as initialization, which only needs a small dataset BSD500-train [Martin et al. 2001] containing 200 images, for optical model fine-tuning. Other competing methods, including JD^2 [Xing and Egiazarian 2021] and DeepOptics-UNet [Metzler et al. 2020], are trained on a larger dataset DIV2K [Agustsson and Timofte 2017] containing roughly 1K images with over 2K resolution since they are trained from scratch. We believe this data/training efficiency is one of the big advantages of ∇-Prox. Additionally, we provide more visual comparisons in Figure III.

H.2 Image Deraining

We perform algorithm unrolling of proximal gradient descent for image deraining. The training data of all methods is made up of 11,200 clean-rain image pairs in Rain14000 [Fu et al. 2017], 1,800 image pairs in Rain1800 [Yang et al. 2017], 700 image pairs in Rain800 [Zhang et al. 2019] and 12 image pairs in Rain12 [Hyun Kim et al. 2013], following MPRNet [Zamir et al. 2021]. We train the solver with a learnable linear operator and an unrolled prior for 250 epochs. As mentioned in the main paper, with a learned solver and an initializer, we could further improve the performance by combining them. This is achieved by another 30 epochs of fine-tuning. The additional quantitative results on Rain100L and Test2800 are shown in Table I, and the additional visual results are shown in Figure IV.

For the network structures of the linear operators, we employ one residual block for the forward routine and another residual block [He et al. 2016] for the adjoint routine. The residual block itself is composed of three convolution blocks with two for feature extraction and one for skip connection.

Table I. Quantitative results (PSNR and SSIM) of image deraining. The best and second-best scores are **highlighted** and underlined.

Method	Rain100H		Rain100L		Test1200		Test2800	
	PSNR	SSIM	PSNR	SSIM	PSNR	SSIM	PSNR	SSIM
PreNet [Ren et al. 2019]	26.77	0.858	32.44	0.950	31.36	0.911	31.75	0.916
MPRNet [Zamir et al. 2021]	30.41	0.890	36.40	0.965	32.91	0.916	33.64	0.938
DGUNet [Mou et al. 2022]	30.66	0.891	38.25	0.974	33.23	0.920	34.01	0.942
Restormer [Zamir et al. 2022]	31.46	0.904	38.99	0.978	33.19	0.926	34.18	0.944
MHNet [Gao and Dang 2023]	30.34	0.903	**39.47**	**0.984**	**33.42**	0.924	-	-
∇-Prox (Learn LinOp)	31.08	0.897	38.09	0.973	32.95	0.913	33.93	0.941
∇-Prox (Learn LinOp + Initializer)	**31.62**	**0.905**	39.06	0.978	33.25	**0.926**	**34.19**	**0.944**

H.3 Compressive Magnetic Resonance Imaging

As we demonstrated in the main paper, we experiment with a variety of solvers for CS-MRI, covering all the techniques that ∇-Prox provides. For the PnP solver, we run all the cases with 24 iterations in total following

[Zhang et al. 2021] and the parameters are manually tuned optimally case by case. For learnable solvers, *i.e.*, unrolled, DEQ, and RL solvers, we train them with a dataset provided in [Wei et al. 2020], which is a resized version (128×128) of PASCAL VOC [Everingham et al. 2015]. Specifically, for unrolled solvers, we start from a PnP solver with a pretrained deep denoising prior and then fine-tune the prior as well as the algorithm parameters by unrolling 10 optimization iterations. For both unrolled and DEQ solvers, we utilize a batch size of 32 and AdamW [Loshchilov and Hutter 2017] optimizer with a constant learning rate of 1×10^{-4}. For RL solvers, the batch size for training is set to 32 and it utilizes an action pack of 5 (predicting parameters of 5 optimization iterations together) and maximum iteration steps of 30, following [Wei et al. 2020]. The additional visual results are shown in Figure V.

H.4 Fast Prototyping Experiments

Here, we provide more experimental setup details for the fast prototyping experiments mentioned in Section 5.5 of the main paper.

Effect of Solver Parameter Setting Strategies. In this experiment, we compare three parameter setting strategies, including fixed, log descent, and our reinforcement-learning-based strategies, on five different tasks, including image deblurring, demosaicing, inpainting, single image super-resolution (SISR), and hyperspectral compressive sensing (HCS). For hyperspectral compressive sensing, we randomly select 5 images from ICVL [Arad and Ben-Shahar 2016] which are cropped to 512×512 for testing. We use Set14 [Zeyde et al. 2012] dataset without cropping for evaluating other tasks. For the parameter setting, there are two parameters to be set for ADMM, *i.e.*, penalty strength ρ and denoising strength σ. They are correlated by $\rho = \lambda\sigma^2$ where λ is the strength of regularize. Following [Zhang et al. 2021], we set $\lambda = 0.23$ as constant and only search the best parameter for σ and derive the best ρ from that. The best σ for fixed strategy is obtained by brute-force search in the range of $[0, 70]$. The best upper and lower bound of σ for the log decent strategy is obtained by brute-force search in the range of $[0, 70]$ as well. For RL strategy, we train the policy network with ICVL where 50 images are randomly selected and cropped to 64×64 (for compressive sensing task), and BSD500-train [Martin et al. 2001] where each image is cropped to 128×128 (for other tasks). The policy network is trained to predict a pair of parameters for every 6 iterations. The maximum number of iterations for all strategies is set to 30. As for the degradation, the deblurring uses Gaussian blur with a width of 15 and sigma of 5, the SISR uses a scaling factor of 2, and Gaussian downsampling with the same blur kernel as the deblurring task. The inpainting randomly masks 50% of pixels. The compressive sensing adopts CASSI [Wagadarikar et al. 2009] system.

Effect of Different Proximal Algorithm Choices. For demosaicing and deblurring, the dataset and training setup for the RL-based parameter scheduler are the same as the first prototyping experiment. For CSMRI, the setup is the same as the experiments of the main paper.

Effect of Regularizers. The dataset/training setup for the RL-based parameter scheduler is the same as the first prototyping experiment. All the different objectives with different regularizers share the same setup.

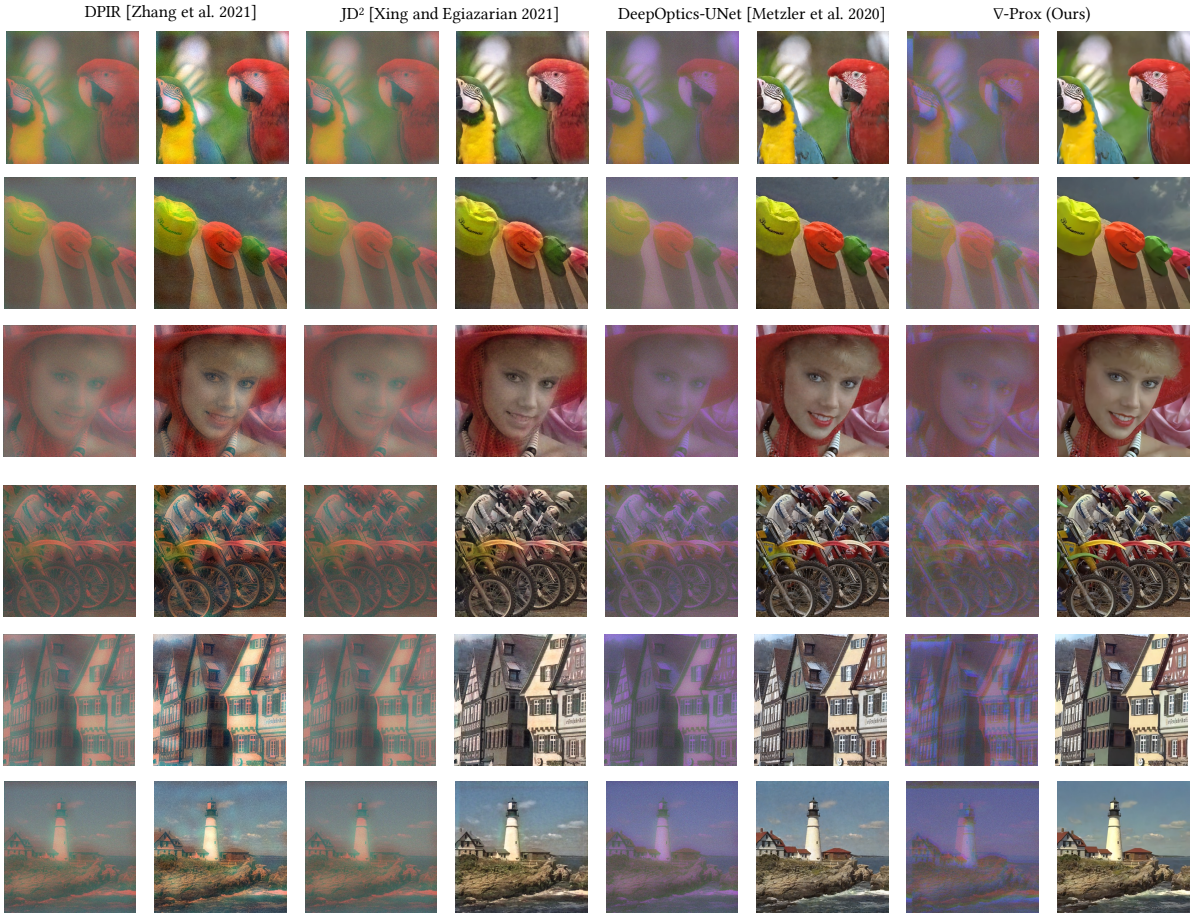


Fig. III. Additional visual results for end-to-end computational optics.

Fig. IV. Additional visual results for image deraining.

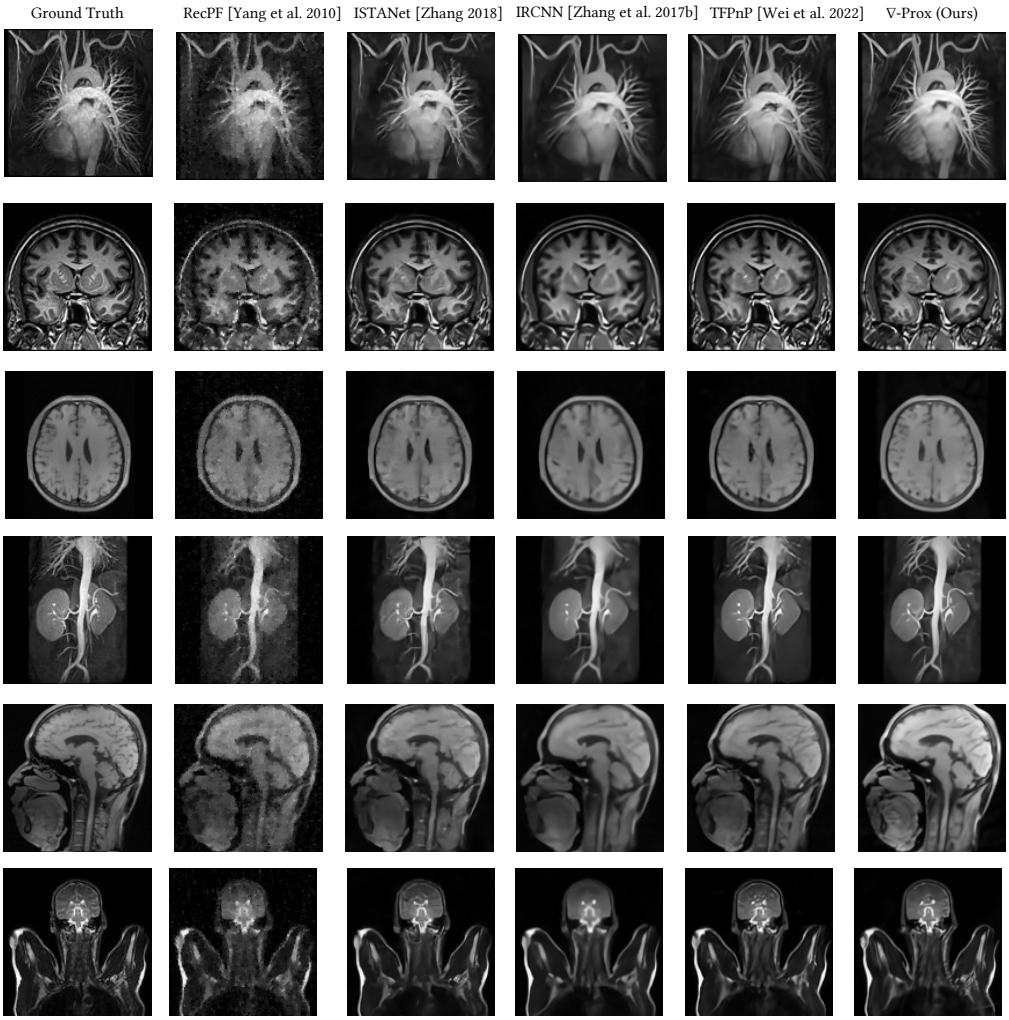


Fig. V. Additional visual results for CSMRI reconstruction.

REFERENCES

Eirikur Agustsson and Radu Timofte. 2017. NTIRE 2017 Challenge on Single Image Super-Resolution: Dataset and Study. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*.

Boaz Arad and Ohad Ben-Shahar. 2016. Sparse Recovery of Hyperspectral Signal from Natural RGB Images. In *European Conference on Computer Vision*. Springer, 19–34.

Shaojie Bai, J Zico Kolter, and Vladlen Koltun. 2019. Deep equilibrium models. *Advances in Neural Information Processing Systems* 32 (2019).

Stephen Boyd, Neal Parikh, Eric Chu, Borja Peleato, Jonathan Eckstein, et al. 2011. Distributed optimization and statistical learning via the alternating direction method of multipliers. *Foundations and Trends® in Machine learning* 3, 1 (2011), 1–122.

Ronald E Bruck Jr. 1975. An iterative solution of a variational inequality for certain monotone operators in Hilbert space. *Bull. Amer. Math. Soc.* 81, 5 (1975), 890–892.

Antonin Chambolle and Thomas Pock. 2011. A first-order primal-dual algorithm for convex problems with applications to imaging. *Journal of mathematical imaging and vision* 40, 1 (2011), 120–145.

Stanley H Chan, Xiran Wang, and Omar A Elgendy. 2016. Plug-and-play ADMM for image restoration: Fixed-point convergence and applications. *IEEE Transactions on Computational Imaging* 3, 1 (2016), 84–98.

Mark Everingham, SM Ali Eslami, Luc Van Gool, Christopher KI Williams, John Winn, and Andrew Zisserman. 2015. The pascal visual object classes challenge: A retrospective. *International journal of computer vision* 111 (2015), 98–136.

Vincent François-Lavet, Peter Henderson, Riashat Islam, Marc G Bellemare, Joelle Pineau, et al. 2018. An introduction to deep reinforcement learning. *Foundations and Trends® in Machine Learning* 11, 3-4 (2018), 219–354.

Xueyang Fu, Jiabin Huang, Delu Zeng, Yue Huang, Xinghao Ding, and John Paisley. 2017. Removing rain from single images via a deep detail network. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 3855–3863.

Hu Gao and Depeng Dang. 2023. Mixed Hierarchy Network for Image Restoration. *arXiv preprint arXiv:2302.09554* (2023).

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

Felix Heide, Steven Diamond, Matthias Nießner, Jonathan Ragan-Kelley, Wolfgang Heidrich, and Gordon Wetzstein. 2016. Proximal: Efficient image optimization using proximal algorithms. *ACM Transactions on Graphics (TOG)* 35, 4 (2016), 1–15.

Tae Hyun Kim, Byeongjoo Ahn, and Kyoung Mu Lee. 2013. Dynamic scene deblurring. In *Proceedings of the IEEE international conference on computer vision*. 3160–3167.

Michael Kruse. 2021. Loop Transformations using Clang's Abstract Syntax Tree. In *50th International Conference on Parallel Processing Workshop*. 1–7.

Timothy Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. 2016. Continuous control with deep reinforcement learning. *International Conference on Learning Representations* (2016).

Longji Lin. 1992. Self-Improving Reactive Agents Based on Reinforcement Learning, Planning and Teaching. *Machine Learning* 8, 3 (1992), 293–321.

Ilya Loshchilov and Frank Hutter. 2017. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101* (2017).

D. Martin, C. Fowlkes, D. Tal, and J. Malik. 2001. A Database of Human Segmented Natural Images and its Application to Evaluating Segmentation Algorithms and Measuring Ecological Statistics. In *Proc. 8th Int'l Conf. Computer Vision*, Vol. 2. 416–423.

Christopher A Metzler, Hayato Ikoma, Yifan Peng, and Gordon Wetzstein. 2020. Deep optics for single-shot high-dynamic-range imaging. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 1375–1385.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. 2013. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602* (2013).

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. 2015. Human-level control through deep reinforcement learning. *Nature* 518, 7540 (2015), 529–533.

Jean-Jacques Moreau. 1965. Proximité et dualité dans un espace hilbertien. *Bulletin de la Société mathématique de France* 93 (1965), 273–299.

Chong Mou, Qian Wang, and Jian Zhang. 2022. Deep Generalized Unfolding Networks for Image Restoration. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 17399–17410.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019).

Jan Peters and Stefan Schaal. 2006. Policy Gradient Methods for Robotics. *International Conference on Intelligent Robots and Systems* (2006), 2219–2225.

Dongwei Ren, Wangmeng Zuo, Qinghua Hu, Pengfei Zhu, and Deyu Meng. 2019. Progressive image deraining networks: A better and simpler baseline. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 3937–3946.

David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. 2014. Deterministic policy gradient algorithms. In *International conference on machine learning*. PMLR, 387–395.

Richard Sutton, David Mcallester, Satinder Singh, and Yishay Mansour. 2000. Policy Gradient Methods for Reinforcement Learning with Function Approximation. *Advances in Neural Information Processing Systems* (2000).

Richard S Sutton and Andrew G Barto. 2018. Reinforcement learning: an introduction. (2018).

Ashwin A Wagadarikar, Nikos P Pitsianis, Xiaobai Sun, and David J Brady. 2009. Video rate spectral imaging using a coded aperture snapshot spectral imager. *Optics express* 17, 8 (2009), 6368–6388.

Kaixuan Wei, Angelica Aviles-Rivero, Jingwei Liang, Ying Fu, Hua Huang, and Carola-Bibiane Schönlieb. 2022. TFPNP: Tuning-free plug-and-play proximal algorithms with applications to inverse imaging problems. *Journal of Machine Learning Research* 23, 16 (2022), 1–48.

Kaixuan Wei, Angelica Aviles-Rivero, Jingwei Liang, Ying Fu, Carola-Bibiane Schönlieb, and Hua Huang. 2020. Tuning-free plug-and-play proximal algorithm for inverse imaging problems. In *International Conference on Machine Learning*. PMLR, 10158–10169.

David S Wile. 1997. Abstract syntax from concrete syntax. In *Proceedings of the 19th international conference on Software engineering*. 472–480.

Wenzhu Xing and Karen Egiazarian. 2021. End-to-end learning for joint image demosaicing, denoising and super-resolution. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 3507–3516.

Wenhan Yang, Robby T Tan, Jiashi Feng, Jiaying Liu, Zongming Guo, and Shuicheng Yan. 2017. Deep joint rain detection and removal from a single image. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 1357–1366.

Syed Waqas Zamir, Aditya Arora, Salman Khan, Munawar Hayat, Fahad Shahbaz Khan, and Ming-Hsuan Yang. 2022. Restormer: Efficient transformer for high-resolution image restoration. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 5728–5739.

Syed Waqas Zamir, Aditya Arora, Salman Khan, Munawar Hayat, Fahad Shahbaz Khan, Ming-Hsuan Yang, and Ling Shao. 2021. Multi-stage progressive image restoration. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 14821–14831.

Roman Zeyde, Michael Elad, and Matan Protter. 2012. On single image scale-up using sparse-representations. In *Curves and Surfaces: 7th International Conference, Avignon, France, June 24-30, 2010, Revised Selected Papers 7*. Springer, 711–730.

He Zhang, Vishwanath Sindagi, and Vishal M Patel. 2019. Image de-raining using a conditional generative adversarial network. *IEEE transactions on circuits and systems for video technology* 30, 11 (2019), 3943–3956.

Kai Zhang, Yawei Li, Wangmeng Zuo, Lei Zhang, Luc Van Gool, and Radu Timofte. 2021. Plug-and-play image restoration with deep denoiser prior. *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2021).

Kai Zhang, Wangmeng Zuo, and Lei Zhang. 2018. FFDNet: Toward a fast and flexible solution for CNN-based image denoising. *IEEE Transactions on Image Processing* 27, 9 (2018), 4608–4622.