Neural Fixed-Point Acceleration for Convex Optimization

Abstract

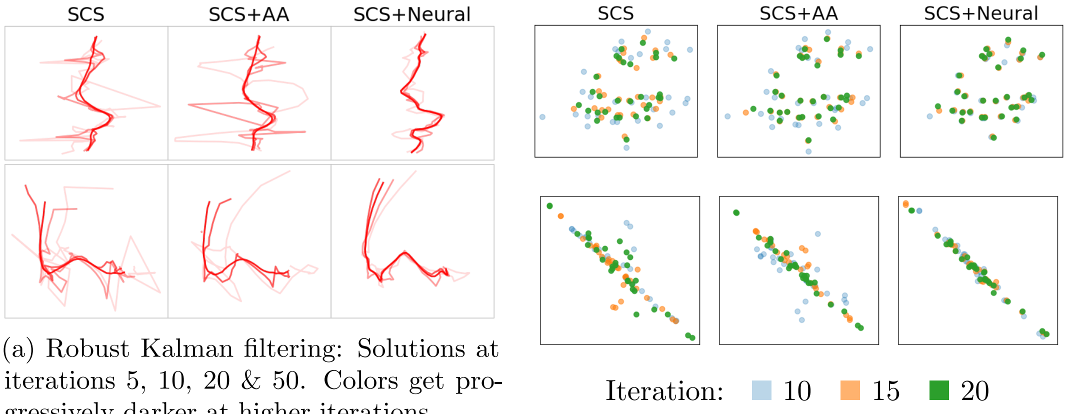
Fixed-point iterations are at the heart of numerical computing and are often a computational bottleneck in real-time applications, which typically instead need a fast solution of moderate accuracy. Classical acceleration methods for fixed-point problems focus on designing algorithms with theoretical guarantees that apply to *any* fixed-point problem. We present *neural fixed-point acceleration*, a framework to automatically learn to accelerate convex fixed-point problems that are drawn from a distribution, using ideas from meta-learning and classical acceleration algorithms. We apply our framework to SCS, the state-of-the-art solver for convex cone programming, and design models and loss functions to overcome the challenges of learning over unrolled optimization and acceleration instabilities. Our work brings neural acceleration into any optimization problem expressible with CVXPY. This is relevant to AutoML as we (meta-)learn improvements to a convex optimization solver that replaces an acceleration component that is traditionally hand-crafted.

1. Introduction

Continuous fixed-point problems are a computational primitive in numerical computing, optimization, machine learning, and the natural and social sciences. Given a map $f : \mathbb{R}^n \to \mathbb{R}^n$, a *fixed point* $x \in \mathbb{R}^n$ is where $f(x) = x$. *Fixed-point iterations* repeatedly apply f until the solution is reached and provably converge under assumptions of f. Most solutions to optimization problems can be seen as finding a fixed point mapping of the iterates, *e.g.* in the convex setting, f could step a primal-dual iterate closer to the KKT optimality conditions of the problem, which remains fixed once it is reached. Recently in the machine learning community, fixed point computations have been brought into the modeling pipeline through the use of differentiable convex optimization (Domke, 2012; Gould et al., 2016; Amos and Kolter, 2017; Agrawal et al., 2019; Lee et al., 2019), differentiable control (Amos et al., 2018), deep equilibrium models (Bai et al., 2019, 2020), and sinkhorn iterations (Mena et al., 2018).

Fixed-point computations are often a computational bottleneck in the larger systems they are a part of. *Accelerating* (*i.e.* speeding up) fixed point computations is an active area of optimization research that involves using the knowledge of prior iterates to improve the future ones. These improve over standard fixed-point iterations but are classically done without learning. The optimization community has traditionally not explored learned solvers because of the lack of theoretical guarantees on learned solvers. For many real-time applications, though, traditional fixed-point solvers can be too slow; instead we need a fast low-accuracy solution. Further, fixed-point problems repeatedly solved in an application typically share a lot of structure and so an application naturally induces a distribution of fixed-point *problem instances*. This raises the question: can we learn a fast and sufficiently-accurate fixed-point solver, when the problem instances are drawn from a fixed distribution?

In this paper, we explore the problem of learning to accelerate fixed-point problem instances drawn from a distribution, which we term *neural fixed-point acceleration*. We focus



(a) Robust Kalman filtering: Solutions at iterations 5, 10, 20 & 50. Colors get progressively darker at higher iterations.

(b) Robust PCA: Iterations 10, 15 & 20.

Figure 1: Visualizing neural-accelerated test instances for robust Kalman filtering and robust PCA. Each line is a single instance. Neural-accelerated SCS quickly stabilizes to a solution, while the SCS and SCS+AA iterations exhibit higher variance.

on convex optimization to ground our work in real applications, including real-time ones such as Tedrake et al. (2010); Mattingley and Boyd (2010). We design a framework for our problem based on *learning to optimize*, *i.e.*, meta-learning (Sect. 2): we learn a model that accelerates the fixed-point computations on a fixed distribution of problems, by repeatedly backpropagating through their unrolled computations. We build on ideas from classical acceleration: we learn a model that uses the prior iterates to improve them, for problems in this distribution. Our framework also captures classical acceleration methods as an instance.

We show how we can learn an acceleration model for convex cone programming with this framework. We focus on SCS (O'Donoghue et al., 2016), which is the state-of-the-art default cone solver in CVXPY (Diamond and Boyd, 2016). However, learning to optimize and acceleration are notoriously hard problems with instabilities and poor solutions, so there are challenges in applying our framework to SCS, which has complex fixed-point computations and interdependencies. Through careful design of models and loss functions, we address the challenges of differentiating through unrolled SCS computations and the subtleties of interweaving model updates with iterate history. Our experiments show that we consistently accelerate SCS in three applications – lasso, robust PCA and robust Kalman filtering.

2. Related work

Learned optimizers and meta-learning. The machine learning community has recently explored many approaches to learning to improve the solutions to optimization problems. These applications have wide-ranging applications, *e.g.* in optimal power flow (Baker, 2020; Donti et al., 2021), combinatorial optimization Khalil et al. (2016); Dai et al. (2017); Nair et al. (2020); Bengio et al. (2020), and differential equations (Li et al., 2020; Poli et al., 2020; Kochkov et al., 2021). The meta-learning and learning to optimize literature, *e.g.* (Li and Malik, 2016; Finn et al., 2017; Wichrowska et al., 2017; Andrychowicz et al., 2016; Metz et al., 2019, 2021; Gregor and LeCun, 2010), focuses on learning better solutions to

Algorithm 1 Neural fixed-point acceleration augments standard fixed-point computations with a learned initialization and updates to the iterates.

```
Inputs: Context $\phi$, parameters $\theta$, and fixed-point map $f$.
$[x_1, h_1] = g_\theta^{\rm init}(\phi)$                      ▷ Initial hidden state and iterate
for fixed-point iteration $t = 1..T$ do
    $\tilde{x}_{t+1} = f(x_t; \phi)$                            ▷ Original fixed-point iteration
    $x_{t+1}, h_{t+1} = g_\theta^{\rm acc}(x_t, \tilde{x}_{t+1}, h_t)$    ▷ Acceleration
end for
```

parameter learning problems that arise for machine learning tasks. Our work is the most strongly connected to the learning to optimize work here and can be seen as an application of these methods to fixed-point computations and convex cone programming.

Fixed-point problems and acceleration. Numerical methods for accelerating fixed-point computations date back decades. Standout classical approaches include *Anderson Acceleration* (AA) (Anderson, 1965) and *Broyden's method* (Broyden, 1965), or variations thereof such as Walker and Ni (2011); Zhang et al. (2020). SCS uses AA by default, and DEQs (Bai et al., 2020) use Broyden's method.

3. Neural fixed-point acceleration

3.1 Problem formulation

We are interested in settings and systems that involve solving a known distribution over fixed-point problems. Each fixed-point problem depends on a *context* $\phi \in \mathbb{R}^m$ that we have a distribution over $\mathcal{P}(\phi)$. The distribution $\mathcal{P}(\phi)$ induces a distribution over fixed-point problems $f(x; \phi) = x$ with a fixed-point map f that depends on the context. Informally, our objective will be to solve this class of fixed-point problems as fast as possible. Notationally, other settings refer to ϕ as a "parameter" or "conditioning variable", but here we will consistently use "context." We next consider a general solver for fixed-point problems that captures classical acceleration methods as an instance, and can also be parameterized with some θ and learned to go beyond classical solvers. Given a fixed context ϕ, we solve the fixed-point problem with alg. 1. At each time step t we maintain the *fixed-point iterations* x_t and a *hidden state* h_t. The *initializer* $g_\theta^{\rm init}$ depends on the context ϕ provides the starting iterate and hidden state and the *acceleration* $g_\theta^{\rm acc}$ updates the iterate after observing the application of the fixed-point map f.

Proposition 1 *Alg. 1 captures Anderson Acceleration as stated* e.g., *in Zhang et al. (2020).*

3.2 Modeling and optimization

We first parameterize the models behind the fixed-point updates in Alg. 1. In neural acceleration, we will use learned models for $g_\theta^{\rm init}$ and $g_\theta^{\rm acc}$. We experimentally found that we achieve good results a standard MLP for $g_\theta^{\rm init}$ and a recurrent model such as an LSTM (Hochreiter and Schmidhuber, 1997) or GRU (Cho et al., 2014) for $g_\theta^{\rm acc}$. While the appropriate models vary by application, a recurrent structure is a particularly good fit as it encapsulates the history of iterates in the hidden state, and uses that to predict a future iterate.

Next, we define and optimize an objective for learning that characterizes how well the fixed-point iterations are solved. Here, we use the *fixed-point residual norms* defined by $\mathcal{R}(x;\phi) := ||x - f(x;\phi)||_2$. This is a natural choice for the objective as the convergence analysis of classical acceleration methods are built around the fixed-point residual. Our learning objective is thus to find the parameters to minimize the fixed-point residual norms in every iteration across the distribution of fixed-point problem instances, *i.e.*

$$\underset{\theta}{\text{minimize}}\ \mathbb{E}_{\phi\sim\mathcal{P}(\phi)} \sum_{t<T} \mathcal{R}(x_t;\phi)/\mathcal{R}_0(\phi), \tag{1}$$

where T is the maximum number of iterations to apply and $\mathcal{R}_0$ is an optional normalization factor that is useful when the fixed-point residuals have significantly different magnitudes. We optimize eq. (1) with a gradient-based method such as Adam (Kingma and Ba, 2014). We assume that the fixed-point map $f(x)$ is differentiable, *i.e.* that we can compute $\nabla f(x)$.

4. Accelerating Convex Cone Programming

We have added neural acceleration to SCS (*Neural SCS*) and integrated it with CVXPY. SCS uses fixed-point iterations to solve cone programs in standard form:

$$\text{minimize}\ c^T x \ \ \text{subject to}\ \ Ax + s = b \quad (x, s) \in \mathbb{R}^n \times \mathcal{K} \tag{2}$$

where $x \in \mathbb{R}^n$ is the primal variable, $s \in \mathbb{R}^m$ is the primal slack variable, $y \in \mathbb{R}^m$ is the dual variable, and $r \in \mathbb{R}^n$ is the dual residual. The set $\mathcal{K} \in \mathbb{R}^m$ is a non-empty convex cone.

4.1 Designing Neural SCS

We now describe how we design Neural SCS as a realization of Alg. 1 in three key steps: modeling, differentiating through SCS, and designing the objective.
(1) Modeling. The input parameters θ come from the initializations of the neural networks that we train, g_θ^{init} and g_θ^{acc}. To construct the input context ϕ for a problem instance, we convert the problem instance into its standard form (eq. (2)), and use the quantities A, b and c, *i.e.* $\phi = [v(A); b; c]$ where $v : \mathbb{R}^{m\times n} \to \mathbb{R}^{mn}$ vectorizes the matrix A. We use an MLP for g_θ^{init}, and a multi-layer LSTM or GRU for g_θ^{acc}.
(2) Differentiating through SCS. The core fixed-point iteration in SCS involves alternating two key steps: (1) projecting current iterates into an affine subspace by solving a linear system; (2) projecting the iterates onto the cone. We thus need to differentiate through both these projections:
(2a) Linear System Solve. We use implicit differentiation, *e.g.* as described in Barron and Poole (2016). Further, for differentiating through SCS, for a linear system $Qu = v$, we only need to obtain the derivative $\frac{\partial u}{\partial v}$, since the fixed-point computation repeatedly solves linear systems with the same Q, but different v. This also lets us use an LU decomposition of Q to speed up the computation of the original linear system solve and its derivative. *(2b) Cone Projections.* We use the cone projection derivative methods developed by Ali et al. (2017); Busseti et al. (2019); we can do so because SCS also formulates the cone program as a homogeneous self-dual embedding (Ye et al., 1994).

Table 1: Sizes of convex cone problems in standard form

	Lasso	*PCA*	*Kalman Filter*
Variables n	102	741	655
Constraints m	204	832	852
nonzeros in A	5204	1191	1652

		Lasso	*PCA*	*Kalman Filter*
Cone dims	Zero	0	90	350
	Non-negative	100	181	100
	PSD	none	[33]	none
	Second-order	[101, 3]	none	[102] + [3]×100

Designing the Objective. The natural choice for the learning objective is the fixed-point residual norm of SCS. With this objective, the interacting algorithmic components of SCS cause g_θ^{acc} and g_θ^{init} to learn poor models for the cone problem. In particular, SCS scales the iterates of feasible problems by τ for better conditioning. However, this causes a serious issue when optimizing the fixed-point residuals: shrinking the iterate-scaling τ artificially decreases the fixed-point residuals, allowing g_θ^{acc} to have a good loss even with poor solutions.

We eliminate this issue in SCS+Neural by normalizing each x_t by its corresponding τ. Thus, the fixed-point residual norm becomes the $||x_t/\tau_t - f(x_t, \phi)/\tau_{f(x_t,\phi)}||$. We are then always measuring the residual norm with $\tau = 1$ for the learning objective. (Note that this change not modify the cone program that we are optimizing.) experiments. [1] In addition, with this objective, we no longer need to learn or predict from τ in the models g_θ^{init} and g_θ^{acc}.

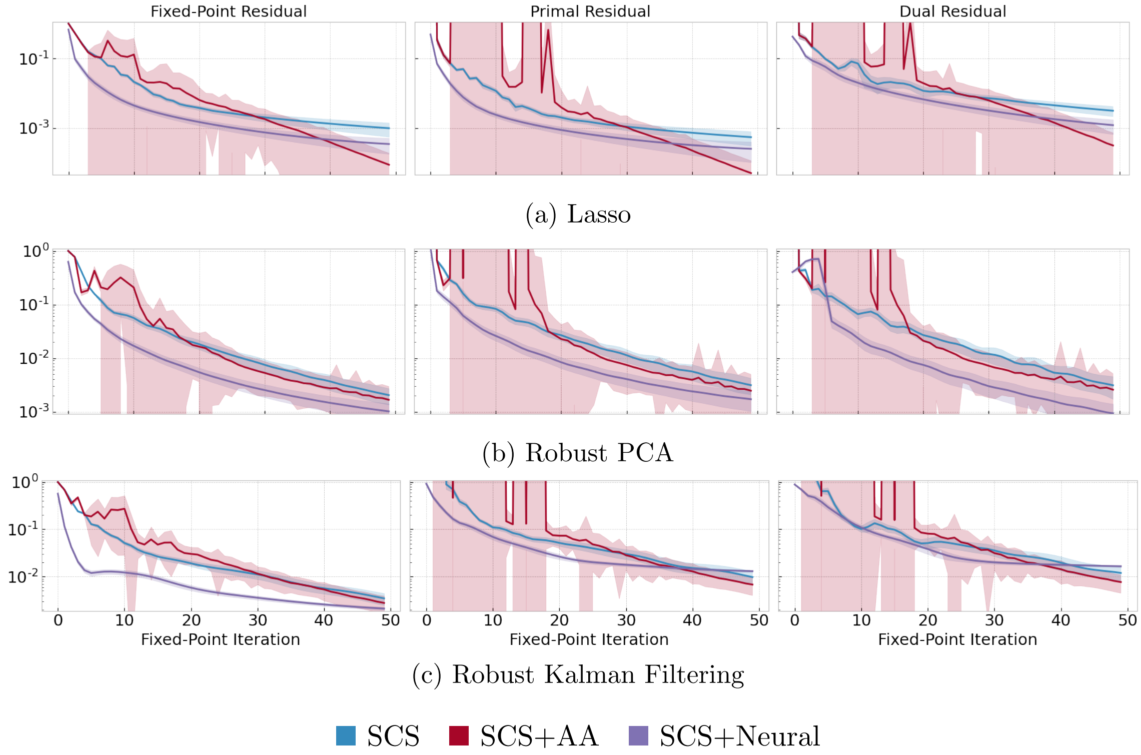
4.2 Experiments

Setup. We demonstrate the experimental performance of SCS+Neural on 3 cone problems: Lasso (Tibshirani (1996)), Robust PCA (Candès et al. (2011)) and Robust Kalman Filtering, chosen similarly to O'Donoghue et al. (2016). Table 1 summarizes problem sizes, types of cones, and cone sizes used in our experiments. We use Adam (Kingma and Ba, 2014) to train for 100,000 model updates. We perform a hyperparameter sweep, and select models with the best validation loss in each problem class. For SCS+AA, we use the default history of 10 iterations. App. C.1.2 describes additional training details.

Results. As an initial proof-of-concept, our experimental results focus on the number of iterations required to achieve required accuracy with SCS+Neural. Figure 2 shows the fixed-point, primal and dual residuals for SCS, SCS+AA, and SCS+Neural. It shows the mean and standard deviation of each residual per iteration, aggregated over all test instances for each solver. SCS+Neural consistently reaches a lower residual much faster than SCS or SCS+AA. e.g., in Lasso (fig. 2a) SCS+Neural reaches a fixed-point residual of 0.001 in 25 iterations, while SCS+AA and SCS take 35 and 50 iterations and SCS respectively. Our improvement for Kalman filtering (fig. 2c) is even higher: we reach a fixed-point residual of 0.01 in 5 iterations, compared to the 30 iterations taken by SCS and SCS+AA. In addition, SCS+AA consistently has high standard deviation, due to its well-known stability issues.

Improving the fixed-point residuals earlier also results in improving the primal/dual residuals earlier. For Robust PCA (fig. 2b), this improvement lasts throughout the 50 iterations. However, SCS+AA has a slight edge in the later iterations for Lasso and Kalman filtering, especially in the primal/dual residuals. These can be improved by adding a regularizer with the final primal-dual residuals to the loss (discussed in App. C.2.2).

[1] (Busseti et al., 2019) also observe this issue for the primal-dual residual map, where they propose a similar solution.



(a) Lasso

(b) Robust PCA

(c) Robust Kalman Filtering

Figure 2: Neural accelerated SCS: Lasso, Robust PCA and Robust Kalman filtering

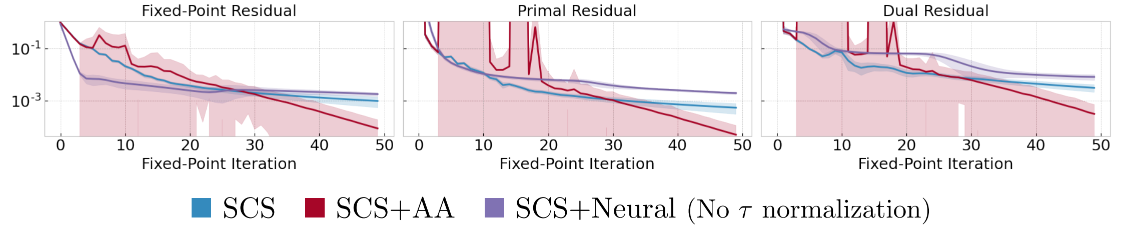


Figure 3: Lasso without τ normalization: a failure mode of neural acceleration (that SCS+Neural overcomes with design).

Importance of τ Normalization in Objective. Figure 3 shows the residuals obtained for Lasso when SCS+Neural does not use τ normalization in the objective. The primal/dual residuals are significantly worse than SCS and SCS+AA. The fixed-point residual shows an initial improvement, but finishes worse. As discussed in Sect. 4, this happens when SCS+Neural achieves a low loss by simply learning a low τ (shown empirically in App. C.2.3).

5. Conclusion and future directions

We have demonstrated the potential of using learning to automatically accelerate a fixed-point solver. As next steps, we are exploring how to learn optimize model that can demonstrate faster wall-clock solution times while scaling to real-world problems. We are also exploring how to apply these ideas to other applications such as motion-planning and optimal transport.

References

Akshay Agrawal, Brandon Amos, Shane Barratt, Stephen Boyd, Steven Diamond, and J Zico Kolter. Differentiable convex optimization layers. In *Advances in Neural Information Processing Systems*, pages 9558–9570, 2019.

Alnur Ali, Eric Wong, and J Zico Kolter. A semismooth newton method for fast, generic convex programming. In *International Conference on Machine Learning*, pages 70–79. PMLR, 2017.

Brandon Amos and J Zico Kolter. Optnet: Differentiable optimization as a layer in neural networks. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 136–145. JMLR. org, 2017.

Brandon Amos, Ivan Jimenez, Jacob Sacks, Byron Boots, and J Zico Kolter. Differentiable mpc for end-to-end planning and control. In *Advances in Neural Information Processing Systems*, pages 8289–8300, 2018.

Donald G Anderson. Iterative procedures for nonlinear integral equations. *Journal of the ACM (JACM)*, 12(4):547–560, 1965.

Marcin Andrychowicz, Misha Denil, Sergio Gomez, Matthew W Hoffman, David Pfau, Tom Schaul, Brendan Shillingford, and Nando De Freitas. Learning to learn by gradient descent by gradient descent. In *Advances in neural information processing systems*, pages 3981–3989, 2016.

Shaojie Bai, J Zico Kolter, and Vladlen Koltun. Deep equilibrium models. *arXiv preprint arXiv:1909.01377*, 2019.

Shaojie Bai, Vladlen Koltun, and J Zico Kolter. Multiscale deep equilibrium models. *arXiv preprint arXiv:2006.08656*, 2020.

Kyri Baker. A learning-boosted quasi-newton method for ac optimal power flow. *arXiv preprint arXiv:2007.06074*, 2020.

Jonathan T. Barron and Ben Poole. The fast bilateral solver, 2016.

Yoshua Bengio, Andrea Lodi, and Antoine Prouvost. Machine learning for combinatorial optimization: a methodological tour d'horizon. *European Journal of Operational Research*, 2020.

Charles G Broyden. A class of methods for solving nonlinear simultaneous equations. *Mathematics of computation*, 19(92):577–593, 1965.

Enzo Busseti, Walaa M Moursi, and Stephen Boyd. Solution refinement at regular points of conic problems. *Computational Optimization and Applications*, 74(3):627–643, 2019.

Emmanuel J. Candès, Xiaodong Li, Yi Ma, and John Wright. Robust principal component analysis? *J. ACM*, 58(3), June 2011. ISSN 0004-5411.

Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

Hanjun Dai, Elias B Khalil, Yuyu Zhang, Bistra Dilkina, and Le Song. Learning combinatorial optimization algorithms over graphs. *arXiv preprint arXiv:1704.01665*, 2017.

Steven Diamond and Stephen Boyd. https://www.cvxpy.org/examples/applications/robust_kalman.html. Accessed: 2020-02-01.

Steven Diamond and Stephen Boyd. CVXPY: A Python-embedded modeling language for convex optimization. *The Journal of Machine Learning Research*, 17(1):2909–2913, 2016.

Justin Domke. Generic methods for optimization-based modeling. In *AISTATS*, volume 22, pages 318–326, 2012.

Priya L. Donti, David Rolnick, and J Zico Kolter. {DC}3: A learning method for optimization with hard constraints. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=V1ZHVxJ6dSS.

Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In *International Conference on Machine Learning*, pages 1126–1135. PMLR, 2017.

Stephen Gould, Basura Fernando, Anoop Cherian, Peter Anderson, Rodrigo Santa Cruz, and Edison Guo. On differentiating parameterized argmin and argmax problems with application to bi-level optimization. 2016.

Karol Gregor and Yann LeCun. Learning fast approximations of sparse coding. In *Proceedings of the 27th international conference on international conference on machine learning*, pages 399–406, 2010.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9 (8):1735–1780, 1997.

Elias Khalil, Pierre Le Bodic, Le Song, George Nemhauser, and Bistra Dilkina. Learning to branch in mixed integer programming. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 30, 2016.

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Dmitrii Kochkov, Jamie A. Smith, Ayya Alieva, Qing Wang, Michael P. Brenner, and Stephan Hoyer. Machine learning accelerated computational fluid dynamics, 2021.

Kwonjoon Lee, Subhransu Maji, Avinash Ravichandran, and Stefano Soatto. Meta-learning with differentiable convex optimization. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 10657–10665, 2019.

Ke Li and Jitendra Malik. Learning to optimize. *arXiv preprint arXiv:1606.01885*, 2016.

Zongyi Li, Nikola Kovachki, Kamyar Azizzadenesheli, Burigede Liu, Kaushik Bhattacharya, Andrew Stuart, and Anima Anandkumar. Fourier neural operator for parametric partial differential equations. *arXiv preprint arXiv:2010.08895*, 2020.

John Mattingley and Stephen Boyd. Real-time convex optimization in signal processing. *IEEE Signal processing magazine*, 27(3):50–61, 2010.

Gonzalo Mena, David Belanger, Scott Linderman, and Jasper Snoek. Learning latent permutations with gumbel-sinkhorn networks. *arXiv preprint arXiv:1802.08665*, 2018.

Luke Metz, Niru Maheswaranathan, Jonathon Shlens, Jascha Sohl-Dickstein, and Ekin D Cubuk. Using learned optimizers to make models robust to input noise. *arXiv preprint arXiv:1906.03367*, 2019.

Luke Metz, C Daniel Freeman, Niru Maheswaranathan, and Jascha Sohl-Dickstein. Training learned optimizers with randomly initialized learned optimizers. *arXiv preprint arXiv:2101.07367*, 2021.

Vinod Nair, Sergey Bartunov, Felix Gimeno, Ingrid von Glehn, Pawel Lichocki, Ivan Lobov, Brendan O'Donoghue, Nicolas Sonnerat, Christian Tjandraatmadja, Pengming Wang, Ravichandra Addanki, Tharindi Hapuarachchi, Thomas Keck, James Keeling, Pushmeet Kohli, Ira Ktena, Yujia Li, Oriol Vinyals, and Yori Zwols. Solving mixed integer programs using neural networks, 2020.

Brendan O'Donoghue, Eric Chu, Neal Parikh, and Stephen Boyd. Conic optimization via operator splitting and homogeneous self-dual embedding. *Journal of Optimization Theory and Applications*, 169(3):1042–1068, 2016.

Michael Poli, Stefano Massaroli, Atsushi Yamashita, Hajime Asama, and Jinkyoo Park. Hypersolvers: Toward fast continuous-depth models. *arXiv preprint arXiv:2007.09601*, 2020.

Russ Tedrake, Ian R Manchester, Mark Tobenkin, and John W Roberts. Lqr-trees: Feedback motion planning via sums-of-squares verification. *The International Journal of Robotics Research*, 29(8):1038–1052, 2010.

Robert Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society. Series B (Methodological)*, 58(1):267–288, 1996.

Homer F Walker and Peng Ni. Anderson acceleration for fixed-point iterations. *SIAM Journal on Numerical Analysis*, 49(4):1715–1735, 2011.

Olga Wichrowska, Niru Maheswaranathan, Matthew W Hoffman, Sergio Gomez Colmenarejo, Misha Denil, Nando Freitas, and Jascha Sohl-Dickstein. Learned optimizers that scale and generalize. In *International Conference on Machine Learning*, pages 3751–3760. PMLR, 2017.

Yinyu Ye, Michael J Todd, and Shinji Mizuno. An $o(\sqrt{nL})$-iteration homogeneous and self-dual linear programming algorithm. *Mathematics of Operations Research*, 19(1):53–67, 1994.

Junzi Zhang, Brendan O'Donoghue, and Stephen Boyd. Globally convergent type-i anderson acceleration for nonsmooth fixed-point iterations. *SIAM Journal on Optimization*, 30(4): 3170–3197, 2020.

Hui Zou and Trevor Hastie. Regularization and variable selection via the elastic net. *Journal of the royal statistical society: series B (statistical methodology)*, 67(2):301–320, 2005.

Appendix A. Batched and Differentiable SCS

In this section, describe our batched and differentiable PyTorch implementation of SCS that enables the neural fixed-point acceleration as a software contribution.

We have implemented SCS in PyTorch, with support for the zero, non-negative, second-order, and positive semi-definite cones. Because our goal is to learn on multiple problem instances, we support batched version of SCS, so that we can solve a number of problem instances simultaneously. For this, we developed custom cone projection operators in PyTorch that allow us to perform batched differentiation.

SCS includes a number of enhancements in order to improve its speed and stability over a wide range of applications. Our implementation in PyTorch supports all enhancements that improve convergence, including scaling the problem data so that it is equilibrated, over-relaxation, and scaling the iterates between each fixed point iteration. Our implementation is thus fully-featured in its ability to achieve convergence using only as many fixed point iterations as SCS.

We are also able to achieve significant improvements in speed through the use of PyTorch JIT and a GPU. However, the focus of this work is on proof-of-concept of neural fixed-point acceleration, and so we have not yet optimized PyTorch-SCS for speed and scale. Our key limitation comes from the necessity of using dense operations in PyTorch, because PyTorch's functionality is primarily centered on dense tensors. While the cone programs are extremely sparse, we are unable to take advantage of its sparsity; this limits the scale of the problems that can be solved. We plan to address these limitations in a future implementation of a differentiable cone solver.

Appendix B. Application: ISTA for elastic net regression

As a first simple application for demonstrating and grounding our fixed-point acceleration, we consider the elastic net regression setting that Zhang et al. (2020) uses to demonstrate the improved convergence of their Anderson Acceleration variant. This setting involves solving elastic net regression (Zou and Hastie, 2005) problems of the form

$$\text{minimize}\ \frac{1}{2}||Ax-b||_2^2 + \mu\left(\frac{1-\beta}{2}||x||_2^2 + \beta||x||_1\right), \tag{3}$$

where $A \in \mathbb{R}^{m\times n}$, $b \in \mathbb{R}^m$. We refer to the objective here as $g(x)$. We solve this with the fixed point computations from the iterative shrinkage-thresholding algorithm

$$f(x) = S_{\alpha\mu/2}\left(x - \alpha\left(A^\top(Ax-b) + \frac{\mu}{2}x\right)\right), \tag{4}$$

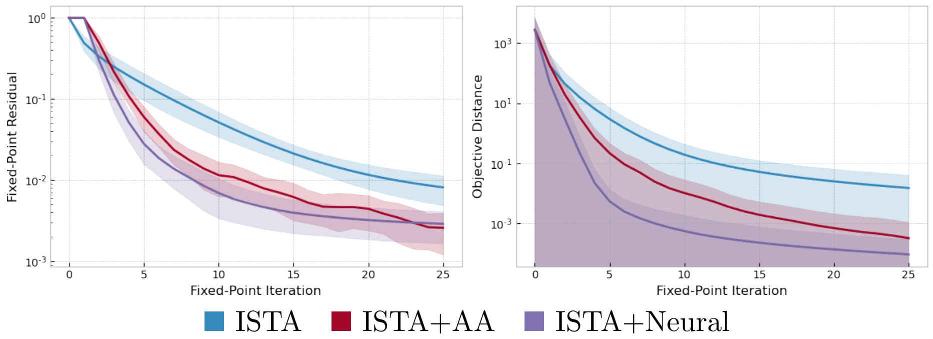


Figure 4: Learning to accelerate ISTA for solving elastic net regression problems. This shows the average fixed-point residual and distance to the optimal objective of a single training run averaged over 50 test samples.

with the shrinkage operator $S_\kappa(x)_i = \text{sign}(x_i)(|x_i| - \kappa)_+$. We follow the hyper-parameters and sampling procedures described in Zhang et al. (2020) and use their Anderson Acceleration with a lookback history of 5 iterations. We set $\mu = 0.001\mu_{\max}$, $\mu_{\max} = ||A^\top b||_\infty$, $\alpha = 1.8/L$, $L = (A^\top A) + \mu/2$, and $\beta = 1/2$. We take $m = n = 25$ and sample A from a Gaussian, $\hat{x}$ from a sparse Gaussian with sparsity 0.1, and generate $b = A\hat{x} + 0.1w$, where w is also sampled from a Gaussian.

We demonstrate in fig. 4 that we competitively accelerate these fixed-point computations. We do this using an MLP for the initialization, GRU for the recurrent unit. In addition to showing the training objective of the normalized fixed-point residuals $\mathcal{R}(x)/\mathcal{R}_0$, we also report the distance from the optimal objective $||g(x) - g(x^\star)||_2^2$, where we obtain $x^\star$ by using SCS to obtain a high-accuracy solution to eq. (4).

Appendix C. Additional Experiments on Neural SCS

C.1 Background and setup

In this section, we provide experimental setup details for results with SCS+Neural. We describe first the different cone problems we use, and then describe additional experimental setup details.

C.1.1 Cone Programs

Lasso. Lasso Tibshirani (1996) is a well-known machine learning problem formulated as follows:

$$\underset{z}{\text{minimize}}\ (1/2)||Fz - g||_2^2 + \mu||z||_1$$

where $z \in \mathbb{R}^p$, and where $F \in \mathbb{R}^{q\times p}$, $g \in \mathbb{R}^p$ and $\mu \in \mathbb{R}_+$ are data. In our experiments, we draw problem instances from the same distributions as O'Donoghue et al. (2016): we generate F as $q \times p$ matrix with entries from $\mathcal{N}(0,1)$; we then generate a sparse vector z^* with entries from $\mathcal{N}(0,1)$, and set a random 90% of its entries to 0; we compute $g = Fz^* + w$, where $w \sim \mathcal{N}(0, 0.1)$; we set $\mu = 0.1||F^T g||_\infty$. We use $p = 100$ and $q = 50$.

Robust PCA. Robust Principal Components Analysis Candès et al. (2011) aims at recovering a low rank matrix of measurements that have been corrupted by sparse noise. It is formulated as:

$$\begin{array}{ll} \text{minimize} & ||L||_* \\ \text{s.t.} & ||S||_1 \leq \mu \\ & L + S = M \end{array}$$

where variable $L \in \mathbb{R}^{p \times q}$ is the original low-rank matrix, variable $S \in \mathbb{R}^{p \times q}$ is the noise matrix, and the data is $M \in \mathbb{R}^{p \times q}$ the matrix of measurements, and $\mu \in \mathbb{R}_+$ that constrains the corrupting noise term.

Again, we draw problem instances from the same distributions as O'Donoghue et al. (2016): we generate a random rank-r matrix L^*, and a random sparse matrix S^* with no more than 10% non-zero entries. We set $\mu = ||S^*||_1$, and $M = L^* + S^*$. We use $p = 30$, $q = 3$ and $r = 2$.

Robust Kalman Filtering. Our third example applies robust Kalman filtering to the problem of tracking a moving vehicle from noisy location data. We follow the modeling of Diamond and Boyd as a linear dynamical system. To describe the problem, we introduce some notation: let $x_t \in \mathbb{R}^n$ denote the state at time $t \in \{0 \ldots T-1\}$, and $y_t \in \mathbb{R}^r$ be the state measurement The dynamics of the system are denoted by matrices: A as the drift matrix, B as the input matrix and C the observation matrix. We also allow for noise $v_t \in \mathbb{R}^r$, and input to the dynamical system $w_t \in \mathbb{R}^m$. With this, the problem model becomes:

$$\begin{array}{lll} \text{minimize} & \Sigma_{t=0}^{N-1}(||w||_2^2 + \mu\psi_\rho(v_t)) & \\ \text{s.t.} & x_{t+1} = Ax_t + Bw_t, & t \in [0 \ldots T-1] \\ & y_t = Cx_t + v_t, & t \in [0 \ldots T-1] \end{array}$$

where our goal is to recover x_t for all t, and where ψ_ρ is the Huber function:

$$\psi_\rho(a) = \begin{cases} ||a||_2 & ||a||_2 \leq \rho \\ 2\rho||a||_2 - \rho^2 & ||a||_2 \geq \rho \end{cases}$$

We set up our dynamics matrices as in Diamond and Boyd, with $n = 50$ and $T = 12$. We generate $w_t^* \sim \mathcal{N}(0, 1)$, and initialize x_0^* to be $\mathbf{0}$, and set μ and ρ both to 2. We also generate noise $v_t^* \sim \mathcal{N}(0, 1)$, but increase v_t^* by a factor of 20 for a randomly selected 20% time intervals. We simulate the system forward in time to obtain x_t^* and y_t for T time steps. Table 1 summarizes the problem instances.

C.1.2 Experimental Setup: Additional Details

For each problem, we create a training set of 100,000 problem instances (50,000 for Kalman filtering), and validation and test sets of 512 problem instances each (500 for Kalman filtering). We allow each problem instance to perform 50 fixed-point iterations for both training and evaluation. We perform a hyperparameter sweep across the parameters of the model, Adam, and training setup, which we detail in Table 2.

Table 2: Parameters used for hyperparameter sweep of SCS+Neural

Adam	
learning rate	$[10^{-4}, 10^{-2}]$
β_1	0.1, 0.5, 0.7, 0.9
v β_2	0.1, 0.5, 0.7, 0.9, 0.99, 0.999
cosine learning rate decay	True, False
Neural Model	
- use initial hidden state	True, False
- use initial iterate	True, False
Initializer:	
- hidden units	128, 256, 512, 1024
- activation function	relu, tanh, elu
- depth	$[0 \dots 4]$
Encoder:	
- hidden units	128, 256, 512, 1024
- activation function	relu, tanh, elu
- depth	$[0 \dots 4]$
Decoder:	
- hidden units	128, 256, 512, 1024
- activation function	relu, tanh, elu
- depth	$[0 \dots 4]$
- weight scaling	[2.0, 128.0]
Recurrent Cell:	
- model	LSTM, GRU
- hidden units	128, 256, 512, 1024
- depth	$[1 \dots 4]$
Misc	
max gradient for clipping	[10.0, 100.0]
batch size	16, 32, 64, 128 [Lasso & PCA]
	5, 10, 25, 50 [Kalman filter]

C.2 Additional Results.

C.2.1 Ablations

We present ablations that highlight the importance of the different pieces of SCS+Neural, using Lasso as a case study.

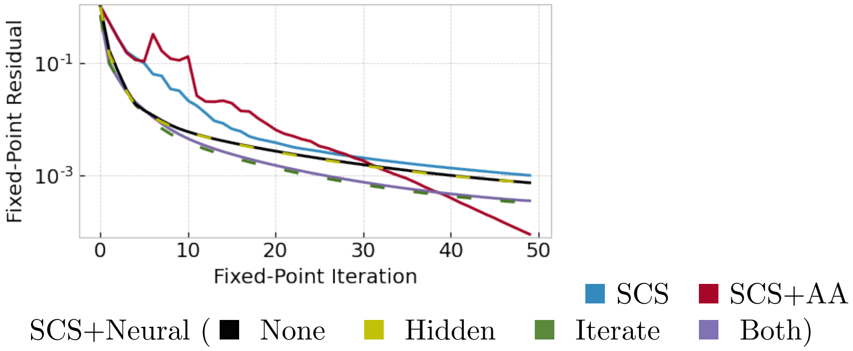


Figure 5: Initializer ablations: Lasso

Initializer. Our first ablation examines the importance of the learned initializer $g_\theta^{\text{init}}(\phi)$ and the initial iterate and hidden state that it provides. We modify g_θ^{init} to output four possibilities: (1) neither initial iterate nor hidden state, (2) only the initial hidden state h_1, (3) only the initial iterate x_1, and (4) both the initial iterate and hidden state $[x_1, h_1]$. Note that in Case (1), the initial context ϕ is not used by the neural acceleration, while Case (4) matches alg. 1.

Figure 5 shows the results for the four cases of g_θ^{init} in SCS+Neural, along with SCS and SCS+AA for comparison. They show the mean of all the test instances per iteration, averaged across three runs with different seeds. First, all four cases of SCS+Neural improve significantly over SCS and SCS+AA in the first 10 iterations, and are near-identical for the first 5-7 iterations. Further, two of the cases, i.e., Case (1) (where g_θ^{init} does not output anything), and Case (2) (where it only outputs h_1) show significantly less improvement than the other two cases; they are also near-identical. In addition, Case (3) (where g_θ^{init} outputs just the initial iterate x_1) is also near-identical to Case (4) (where it outputs both $[x_1, h_1]$). This suggests that g_θ^{init} able to start the fixed-point iteration with a good x_1, while the initial h_1 it has learned does not have much impact.

C.2.2 Regularizing with Primal-Dual Residuals.

We can also optimize other losses beyond the fixed-point residuals to reflect more properties that we want our fixed-point solutions to have. Here we discuss how we can add the primal and dual residuals to the loss, which are different quantities that the fixed-point residuals. The loss is designed to minimize the fixed-point residual as early as possible, so sometimes, we see that the final primal-dual residuals of SCS+Neural are slightly worse than SCS and SCS+AA.

Because the primal/dual residuals also converge under the fixed-point map, we can adapt the loss to include them primal/dual residuals as well, *i.e.*, similar to eq. (1), we can define an updated learning objective:

$$\begin{aligned}\underset{\theta}{\text{minimize}}\ \mathbb{E}_{\phi\sim\mathcal{P}(\phi)}(1-\lambda)\sum_{t<T}\mathcal{R}(x_t;\phi)/\mathcal{R}_0(\phi)&\\ +\lambda||[p(x_T,\phi);d(x_T,\phi)]||_2&\end{aligned}\tag{5}$$

where $\lambda\in[0,1]$ is the regularization parameter, p and d are the primal and dual residuals at x_T. At $\lambda=0$, this is our original objective eq. (1); at $\lambda=1$, this objective ignores the fixed-point residuals and only minimizes the final primal and dual residuals obtained after T iterations. We ablate λ in our experiments.

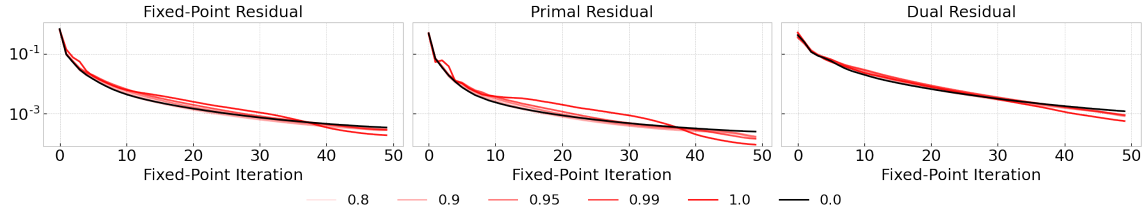


Figure 6: Ablations for regularization with primal & dual residuals: Lasso

Our next ablation examines the impact of regularizing the loss with the final primal/dual residuals. Figure 6 shows all three residuals for SCS+Neural for λ ranging from 0.8 to 1.0, in addition to the original SCS+Neural (with $\lambda=0$). [2] For clarity, we show only the means over all test instances for all seeds; the standard deviations are similar to the earlier Lasso experiments. As λ increases, all three residuals get a little worse than the original SCS+Neural in early iterations, while there is an improvement in all three residuals in the later iterations (past iteration 35). The maximum improvement in the final primal and dual residuals at $\lambda=1$, when the learning objective is to minimize only the final primal/dual residuals. These results suggest that this regularization could be used to provide a flexible tradeoff of the residuals of the final solution for the speed of convergence of the fixed-point iteration.

C.2.3 τ Normalization

We can understand the behavior of g_θ^{acc} by examining how τ changes over the fixed-point iterations. Figure 7 shows the mean and standard deviation of the learned τ values, averaged across all test instances and across runs with all seeds. Note that SCS and SCS+AA quickly find their τ (by iteration 3-4), and deviate very little from it. SCS+Neural, however, starts at a very low τ that slowly increases; this results in very low initial fixed-point residuals (and thus a better loss for g_θ^{acc}), but poor quality solutions with high primal/dual residuals.

C.2.4 Visualizing Convergence

Lastly, we discuss in more detail the visualizations of convergence that we illustrated in Sect. 1. Figure 1a shows the solutions of two different test instances for Robust Kalman

[2] We only focus on high λ because we see only marginal differences from the original SCS+Neural at lower λ.

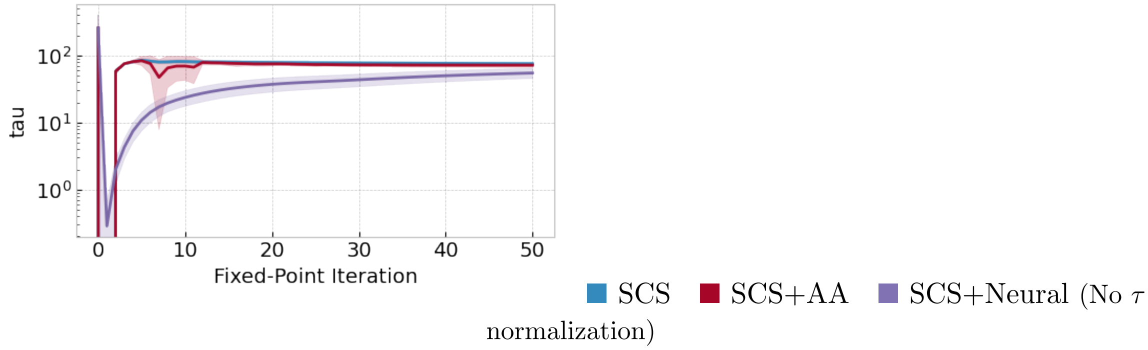


Figure 7: We observe that without τ normalization, a failure mode of neural acceleration is that it learns to produce low τ values that artificially reduce the fixed-point residuals while not nicely solving the optimization problem.

filtering at iterations 5, 10, 20 and 50. Lighter paths show earlier iterations, and darker paths show later iterations. For both instances, e see that SCS+Neural has few visible light (intermediate) paths; most of them are covered by the final dark path, and those that are visible are of the lightest shade. This indicates that SCS+Neural has mostly converged by iteration 5, unlike SCS and SCS+AA, which have many more visible intermediate (light) paths around their final path. Figure 1b shows the solutions for two instances in Robust PCA at iterations 10, 15 and 20 for SCS, SCS+AA and SCS+Neural. It is clear that, for both instances, the SCS+Neural has almost converged by iteration 10. In contrast, SCS and SCS+AA show many more visible distinct points at iterations 10 and 15, indicating they have not yet converged.