Neural Fixed-Point Acceleration for Second-order Cone Optimization Problems

Anonymous authors
Paper under double-blind review

Abstract

Continuous fixed-point problems are a computational primitive in numerical computing, optimization, machine learning, and the natural and social sciences, and have recently been incorporated into deep learning models as optimization layers. Acceleration of fixed-point computations has traditionally been explored in optimization research without the use of learning. In this work, we introduce neural fixed-point acceleration, a framework to automatically learn to accelerate fixed-point problems that are drawn from a distribution; a key question motivating our work is to better understand the characteristics that make neural acceleration more beneficial for some problems than others. We apply the framework to solve second-order cone programs with the Splitting Conic Solver (SCS), and evaluate on distributions of Lasso problems and Kalman filtering problems. Our main results show that we are able to get a $10\times$ performance improvement in accuracy on the Kalman filtering distribution, while those on Lasso are much more modest. We then isolate a few factors that make neural acceleration much more useful on the Kalman filtering distribution than on the Lasso distribution; we apply a number of problem and distribution modifications on a scaled-down version of the Lasso problem, adding in properties that make it structurally closer to Kalman filtering, and show when the problem benefits from neural acceleration.

1 Introduction

Given a map $f : \mathbb{R}^n \to \mathbb{R}^n$, a *fixed point* of f is a point $x \in \mathbb{R}^n$ where $f(x) = x$. *Fixed-point iterations* repeatedly apply f until a fixed point is reached and provably converge under assumptions of f (Giles, 1987). Continuous fixed-point problems are a computational primitive in numerical computing, optimization, machine learning, and the natural and social sciences. Recent work in machine learning has incorporated fixed-point computations into deep learning models as optimization layers, e.g., through differentiable convex optimization (Domke, 2012; Gould et al., 2016; Amos & Kolter, 2017; Agrawal et al., 2019; Lee et al., 2019), differentiable control (Amos et al., 2018), deep equilibrium models (Bai et al., 2019; 2020; 2022), and Sinkhorn iterations (Mena et al., 2018). Because these layers are now solving optimization problems, they become a significant computational bottleneck in training and deploying when using optimization layers. Quickly predicting solutions to these optimization problems would be a significant step in speeding up the use of optimization layers.

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 acceleration methods improve over standard fixed-point iterations but are classically done without learning, because of the lack of theoretical guarantees on learned solvers. However, for many real-time applications, traditional fixed-point solvers can be too slow.

On the other hand, fixed-point problems that get repeatedly solved in an application typically share a lot of structure, e.g., motion planning with noisy observations. Such an application naturally induce a distribution of fixed-point problem instances. This raises the question: can we learn to accelerate a fixed-point solver, when the problems are drawn from a fixed distribution?

In this paper, we study the problem of learning to accelerate fixed-point problem instances drawn from a distribution, which we term *neural fixed-point acceleration*; a key motivation of this work is also to better understand the characteristics that make neural acceleration more beneficial or easier for some problems than others. We design a framework for our problem based on *learning to optimize*, *i.e.*, meta-learning (see Section 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 history of past iterates to predict the next iterate.

Concretely, we first present a neural acceleration framework that learns an acceleration model as a drop-in replacement for a classical acceleration technique such as Anderson Acceleration, i.e., that takes the current and past few fixed-point iterates and predicts the next iterate at every step in the optimization. We then generalize this framework to support models that predict fixed-point iterates only at intermittent steps in the optimization. Predicting at intermittent steps only allows for greater training and inference efficiency than predicting iterates at every step, since we no longer need to backpropagate through the entire history of fixed-point iterations, but instead can use truncated gradients (Wu et al., 2018; Metz et al., 2021).

We apply this framework to the Splitting Conic Solver (O'Donoghue et al., 2016), solving two second-order cone problem distributions: one over Kalman filtering problems and another over Lasso problems. Our main results show that our best acceleration models achieve more than 10$\times$ improvement in accuracy on the Kalman filtering problem distribution, when compared to SCS with Anderson Acceleration; our improvement on the Lasso problem distribution is more modest. Through the use of overparametrization (Arora et al., 2018), we are able to reduce the training time of our most expensive application, Kalman filtering, by a factor of over 1.4.

In the second part of this paper, we isolate a few factors that make neural acceleration much better on the Kalman filtering distribution than the Lasso distribution. We apply a number of distribution and problem modifications to a scaled-down version of our Lasso distribution, adding in properties that make it structurally closer to the Kalman filtering distribution. Our experiments, while only illustrative, provide evidence that the structural differences we identify (i.e., the amount and type of randomness in the problem distribution, the set of cones in the problem representation) are a few of the factors that allow the robust Kalman filtering problem to benefit more from neural acceleration. Our experiments suggest that linear dynamical systems may be a class of optimization problems that benefit from neural acceleration.

2 Related Work

There are many classic numerical methods for acceleration, such as *Anderson Acceleration* (AA) (Anderson, 1965) (also the default solver in SCS), *Broyden's method* (Broyden, 1965), and Walker & Ni (2011); Zhang et al. (2020). Most closely related to our work is that of Bai et al. (2022), who also learn an acceleration model; however, they focus on deep equilibrium models, and their model predicts the initial iterate and AA update coefficients at each step. Our approach is complementary to theirs, as we learn the full iterates at both the initial and the later steps; while this modeling makes it more challenging to accelerate similarly to the AA path, it allows us more easily to capture iterates for accelerating problems that are difficult for AA.

Our work is also related to the learning-to-optimize literature, of which Amos (2022) provides a tutorial. The meta-learning and learning-to-optimize work, *e.g.* (Li & Malik, 2016; Finn et al., 2017; Wichrowska et al., 2017; Andrychowicz et al., 2016; Metz et al., 2019; 2021; Gregor & LeCun, 2010), aim to learn better solutions to problems that arise in machine learning tasks. Bastianello et al. (2021) approximates the fixed-point iteration with the closest contractive fixed-point iteration. Ichnowski et al. (2021) use reinforcement learning to improve quadratic programming solvers. There have also been many recent works on learning-to-optimize that are application-driven, e.g., *e.g.* 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 solving differential equations (Li et al., 2020; Poli et al., 2020; Kochkov et al., 2021).

3 Background

In this section, we introduce the definitions and background necessary for the rest of the paper.

3.1 Convex Cone Optimization

Convex cone programming is a class of optimization problems that are capable of representing *any* convex optimization problem (Nemirovski, 2007). In standard form, conic optimization involves solving the following *primal-dual* problems:

$$\begin{aligned} &\text{minimize } c^T x && \text{maximize } -b^T y \\ &\text{s.t. } Ax + s = b && \text{s.t. } -A^T y + r = c \\ &\quad (x, s) \in \mathbb{R}^n \times \mathcal{K} && \quad (r, y) \in \{0\}^n \times \mathcal{K}^* \end{aligned} \tag{1}$$

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 with dual cone $\mathcal{K}^* \in \mathbb{R}^m$. (We refer the reader to Boyd & Vandenberghe (2004) for an introduction to conic optimization.)

3.2 Solver

We use the Splitting Conic Solver (SCS) (O'Donoghue et al., 2016) as our (differentiable) solver in our neural acceleration framework. SCS is a state-of-the-art fixed-point algorithm that solves eq. (1). In this section, we briefly describe its formulation and fixed-point iterations, as these are necessary to define our metric of convergence, the fixed-point residual.

Formulation. SCS solves conic optimization problems by converting the pair of primal-dual optimization problems into a homogeneous self-dual embedding (Ye et al., 1994), which remains feasible even if the original pair of primal-dual problems is not feasible. We describe this formulation below:

$$\begin{bmatrix} r \\ s \\ \kappa \end{bmatrix} = \begin{bmatrix} 0 & A^T & c \\ -A & 0 & b \\ -c^T & -b^T & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ \tau \end{bmatrix} \tag{2}$$

where x, y, r, s are as in eq. (1) and $\kappa, \tau \in \mathbb{R}_+$, i.e., κ, τ are non-negative scalars.

We use the following notation to simplify eq. (2):

$$u = \begin{bmatrix} x \\ y \\ \tau \end{bmatrix}, v = \begin{bmatrix} r \\ s \\ \kappa \end{bmatrix}, Q = \begin{bmatrix} 0 & A^T & c \\ -A & 0 & b \\ -c^T & -b^T & 0 \end{bmatrix}.$$

The homogeneous self-dual embedding (2) is then:

$$\begin{aligned} &\text{find } (u, v) \\ &\text{s.t. } v = Qu \\ &\quad u, v \in \mathcal{C} \times \mathcal{C}^* \end{aligned} \tag{3}$$

where $\mathcal{C} = \mathbb{R}^n \times \mathcal{K}^* \times \mathbb{R}_+$ is a cone with dual $\mathcal{C}^* = \{0\}^n \times \mathcal{K} \times \mathbb{R}_+$.

Core Algorithm. Let $\Pi_S(x)$ denote the Euclidean projection of x to the subspace S. The core SCS algorithm is given by the following equations:

$$\tilde{u}^{k+1} = (I + Q)^{-1}(u^k + v^k) \tag{4}$$

$$u^{k+1} = \Pi_C(\tilde{u}^{k+1} - v^k) \tag{5}$$

$$v^{k+1} = v^k - \tilde{u}^{k+1} + u^{k+1} \tag{6}$$

The first step (eq. (4)) projects the current iterates into an affine subspace by solving a linear system. The second step projects the resulting iterates onto the cone $\mathcal{C}$. The third step simply updates v^{k+1} with the difference $u^{k+1} - \tilde{u}^{k+1}$. We note that SCS uses $||u^{k+1} - u^k||_2$ as its *fixed-point residual*, since u^i converges to its fixed point at the optimal solution.

3.3 Applications

Lasso. The 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.

We draw problem instances from the same distribution as O'Donoghue et al. (2016): For each instance, we create a matrix $F \in \mathbb{R}^{q \times p}$ where each entry is drawn from $\mathcal{N}(0, 1)$. We create a vector $z^* \in \mathbb{R}^p$, also with each entry in $\mathcal{N}(0, 1)$, 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 $\mu = 0.1||F^T g||_\infty$.

Robust Kalman Filtering. Our second example applies robust Kalman filtering to the problem of tracking a moving vehicle from noisy location data. We follow the modeling of Diamond & Boyd (2022) 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 $t \in \{0 \dots T-1\}$, and $y_t \in \mathbb{R}^r$ be the state measurement. The dynamics of the system are denoted by matrices: F as the drift matrix, G as the input matrix and H 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:

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

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 simulate the system forward in time to obtain x_t^* and y_t for T time steps. Our optimization variables for this problem are thus x_t, w_t and v_t. For ease of reference, we include the full dynamics matrices F, G and H in Appendix A.

4 Neural Acceleration: A First Framework

4.1 Definitions

We introduce some definitions to describe our framework. A fixed-point problem is defined by a *context* $\phi \in \mathcal{R}^m$ drawn from a distribution $\mathcal{P}(\phi)$; we will use f to denote the fixed-point map, and θ to denote learned parameters. Our goal is to find the fixed points of $f(x; \phi)$. We define two models that will be learned:

- the *initializer* g_θ^{init} as the model that provides a starting iterate, typically using as input the initial problem instance context ϕ,
- the *acceleration model* g_θ^{acc} as the model that updates the iterate at all further iterations after observing the application of the fixed-point map f.

4.2 Framework

Next, we describe our first neural fixed-point acceleration framework, shown in Alg. 1. Given a fixed context ϕ, we solve the fixed-point problem as follows. At each time step t we maintain the *fixed-point iterations* x_t and a *hidden state* h_t. In the first time step, the initializer g_θ^{init} takes as input the context ϕ, and provides the starting iterate and the first hidden state. In all further time-steps, the acceleration model g_θ^{acc} uses the hidden state h_t and the current fixed-point iterate x_t, and the fixed-point map f to provide an updated iterate x_{t+1}.

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

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

4.3 Modeling and optimization

4.3.1 Modeling

Model. We use a standard MLP for g_θ^{init} and a recurrent model such as an LSTM (Hochreiter & Schmidhuber, 1997) or GRU (Cho et al., 2014) for g_θ^{acc}. A recurrent model is a natural choice for the acceleration model g_θ^{acc} as it encapsulates the history of iterates in the hidden state, and uses that to predict a future iterate.

We construct the input context ϕ for a problem instance by converting it into its standard form (1), and using A, b, c to define ϕ, i.e., $\phi = [v(A); b; c]$ where $v : \mathbb{R}^{m \times n} \to \mathbb{R}^{mn}$. The parameters θ are initialized through the initialization of g_θ^{init} and g_θ^{acc}.

Loss. To characterize how well the fixed-point iterations are solved, we use the *fixed-point residual norms* defined by $\mathcal{R}(x; \phi) \stackrel{\text{def}}{=} ||x - f(x; \phi)||_2$. This is a natural choice for the objective as the convergence analysis of SCS and 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:

$$\underset{\theta}{\text{minimize}} \ \mathbb{E}_{\phi \sim \mathcal{P}(\phi)} \sum_{t<T} \mathcal{R}(x_t; \phi)/\mathcal{R}_0(\phi), \tag{7}$$

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. (7) with a gradient-based method such as Adam (Kingma & Ba, 2014). For this, we need that the fixed-point map $f(x)$ is differentiable, *i.e.* that we can compute $\nabla f(x)$. In the next section, we describe how to differentiate through the fixed-point map of SCS.

4.3.2 Differentiating through SCS

Recall that 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:

1. *Linear System Solve.* We use implicit differentiation, *e.g.* as described in Barron & 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.
2. *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).

Normalization of the Loss. As described earlier, the natural choice for the learning objective is the fixed-point residual norms of SCS. However, SCS scales the iterates of feasible problems by τ for better conditioning, and this causes a serious issue when optimizing the fixed-point residuals: shrinking the iterate-

Table 1: Sizes of convex cone problems in standard form

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

		Lasso	*Kalman Filter*
Cones	Zero	0	350
	Non-negative	100	100
	Second-order	[101, 3]	[102] + [3]×100

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 does not modify the cone program that we are optimizing.) We show the importance of this design choice by ablating the τ normalization in Appendix D. Busseti et al. (2019) also observe this issue for the primal-dual residual map, where they propose a similar solution. 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}.

4.4 Experiments

We show experimental results on two second-order cone problems: Lasso and Robust Kalman Filtering. For ease of reference, we denote SCS accelerated with Anderson Acceleration as *SCS+AA*, and the learning-augmented SCS as *SCS+Neural*.

4.4.1 Experimental Setup

Problem distributions. We now describe how we instantiate the optimization problems from Section 3.3. For Lasso, we create a training set of 100,000 problem instances and validation and test sets of 512 problems each; we use $p = 50$, $q = 100$. For Kalman filtering, we create a training set of 50,000 problems and validation and test sets of 500 each. We set up our dynamics matrices as in Diamond & Boyd (2022), 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. Table 1 summarizes problem sizes, types of cones, and cone sizes of these problems, obtained through CVXPY's canonicalization (Agrawal et al., 2019).

Training and Evaluation. We use Adam (Kingma & Ba, 2014) to train the g_θ^{init} and g_θ^{acc} for up to 100,000 model updates. To solve a problem instance, we allow it to perform 50 fixed-point iterations for both training and evaluation. For SCS+AA, we use a history of 5 iterations. We perform a hyperparameter sweep across the parameters of the model, Adam, and training setup, and use the best models for the results below. Table 3 shows the values used in the hyperparameter sweep.

4.4.2 Results

Figure 1 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. We see that SCS+consistently reaches a lower residual faster than SCS or SCS+AA, in the earlier iterations, but SCS+AA is able to slightly improve over SCS+Neural in the last few iterations (e.g., past iteration 40). For example, in Lasso (Figure 1a), 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; moreover, improving the fixed-point residuals earlier also results in corresponding improvement in the primal/dual residuals. Our improvement for Kalman filtering (Figure 1b) is more mixed: we reach a fixed-point residual of 0.01 in 5 iterations, compared to the 30 iterations taken by SCS and SCS+AA; however, the primal/dual residuals do not show as much improvement. In addition, SCS+AA consistently has high standard deviation, due to its well-known stability issues.

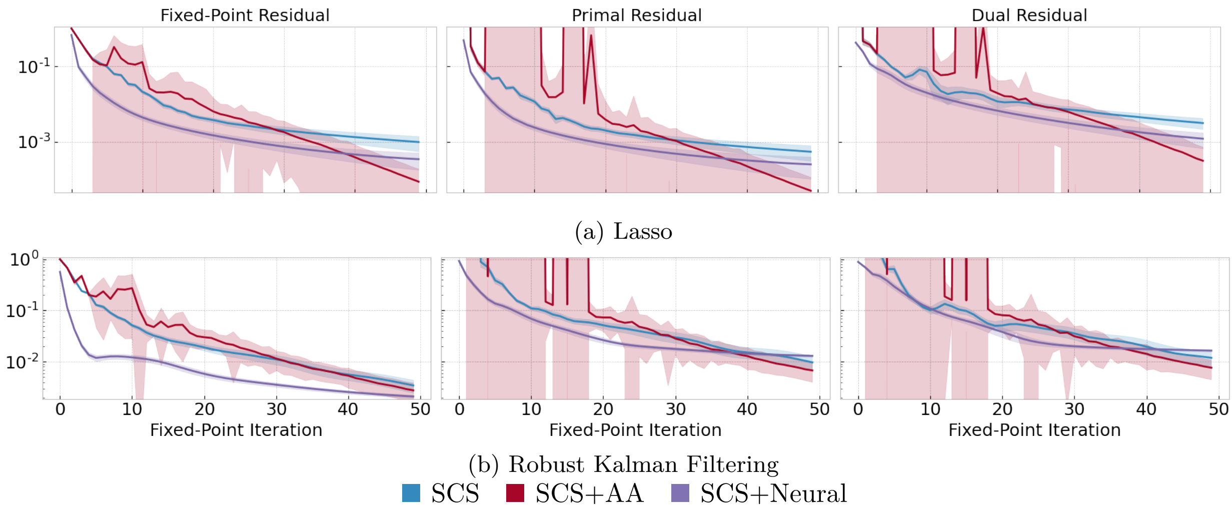


Figure 1: Learned acceleration models in SCS+Neural with recurrent models show modest improvements over SCS and SCS+AA.

5 Neural Acceleration with Intermittent Model Access

We now describe how we generalize the framework of Section 4 to accelerate fixed-point iterations with only intermittent model access. Unrolling through the entire sequence of fixed point iterations is computationally expensive and memory intensive, and is likely the cause of substantially slower training. We instead use truncated gradients (Wu et al., 2018; Metz et al., 2021) to update the acceleration model to improve in the local region around where it was applied. This allows us to learn by accessing the model only at intermittent iterations.

5.1 Framework

We now describe our acceleration framework, followed by the learning algorithm needed to train models appropriately for this framework. Since we are no longer accessing the model at each iteration, we cannot use the hidden state of the recurrent model to encode the current problem and solution state. Instead we will use the history of the iterates and the problem context as input to allow the model to effectively reconstruct the state for a particular iteration.

Acceleration Framework. Formally, let the *access set* A be the set of fixed-point iterations at which the model is accessed. The framework now needs to do the following: (1) apply g_θ^{acc} only at iterations in the access set (we assume g_θ^{init} is always applied); (2) keep the history H_i of the last k iterates at iteration i; (3) g_θ^{acc} now needs to use the iterate history H_i and context ϕ in place of the hidden state to predict the next iterate. The full acceleration framework is presented in Alg. 2.

Learning Algorithm. Next, we describe how we learn an acceleration model with intermittent access using truncated gradients. We first observe that the iterate sequence that the model needs to predict from is likely to change as the model learns. Because these are high-dimensional iterates, we use a strategy motivated by replay buffers in reinforcement learning (Sutton & Barto, 2018): we use our model to generate iterates to learn from, and then use those iterates with the fixed-point map f to improve the model.

We introduce one more definition to describe the learning framework: the *residual interval* r_i is the number of fixed-point iterations performed after the access at iteration i. r_0 is the number of fixed-point iterations performed after the access to g_θ^{init}, and we refer to all residual intervals as $R = \{r_i\}$.

Specifically, our learning algorithm is the following:

Algorithm 2 Neural fixed-point acceleration with intermittent model access

Inputs: Context ϕ, parameters θ, fixed-point map f, access set A, iterate history size k
$x_0 = g_\theta^{\text{init}}(\phi)$ ▷ Initial model-generated iterate
for fixed-point iteration $t = 1..T$ **do**
  $\tilde{x}_t = f(x_{t-1}; \phi)$ ▷ Original fixed-point iteration
  $H_t = \{x_{t-k}, \ldots x_{t-1}\}$ ▷ Update iterate history
  if $t \in A$ **then**
    $x_t = g_\theta^{\text{acc}}(\tilde{x}_t, H_t, \phi)$ ▷ Apply the learned acceleration model
  else
    $x_t = \tilde{x}_t$ ▷ Use same iterate without acceleration
  end if
end for

1. We generate a sequence of iterates $H = [x_1, x_2 \ldots x_t]$ by repeatedly applying f, accelerating an iterate x_i with our current g_θ^{acc} if $i \in A$.
2. We use this sequence of iterates to create our iterate history H_i for iteration $i \in A$.
3. We use g_θ^{acc} to predict new iterates using the current iterate, iterate history, and context: $y_{i+1} = g_\theta^{\text{acc}}(x_i, H_i, \phi)$.
4. We compute fixed-point residuals on y_{i+1} for another r_i steps as the loss for iteration i: $\mathcal{L}_i = \sum_{j \leq r_i} \mathcal{R}(y_{i+j}; \phi)$. Further, for a pair of accesses $i_1, i_2 \in A$, if $r_i > i_2 - i_1$, then fixed-point residuals (and therefore the loss $\mathcal{L}_i$) include a model access for i_2 as well.
5. We compute our total loss $\mathcal{L} = \sum_{i \in A} \mathcal{L}_i$, differentiate with respect to θ, and update θ.

5.2 Modeling

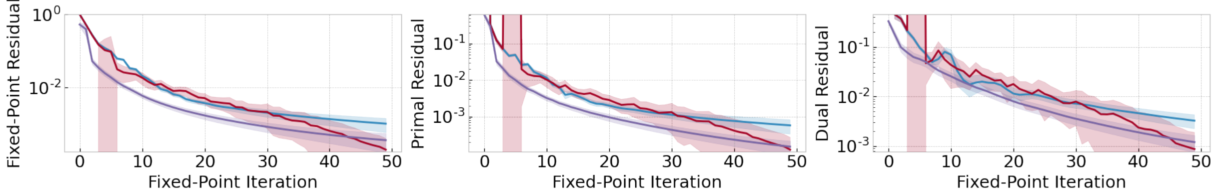
Model Architecture. Throughout, we use two MLPs: one for g_θ^{init}, and one for g_θ^{acc}. We use an MLP for the g_θ^{acc} here, rather than a recurrent model because we found more efficient to train MLPs rather than recurrent models, and because we only train and predict at intermittent fixed-point iterations. Our fixed-point iterates for this model are both iterates of SCS, u and v, which we denote as $(u; v)$. We use both iterates because: (1) our access is intermittent and we do not have a hidden state, so we provide g_θ^{acc} both iterates as the iterate history; (2) predicting both u and v is helpful, since a mismatched v can degrade what might otherwise be a good prediction of u.

As in Section 4, we construct the input context ϕ for a problem instance by converting it into its standard form (1), and using A, b, c to define ϕ, i.e., $\phi = [v(A); b; c]$ where $v : \mathbb{R}^{m \times n} \to \mathbb{R}^{mn}$. The parameters θ are initialized through the initialization of g_θ^{init} and g_θ^{acc}.

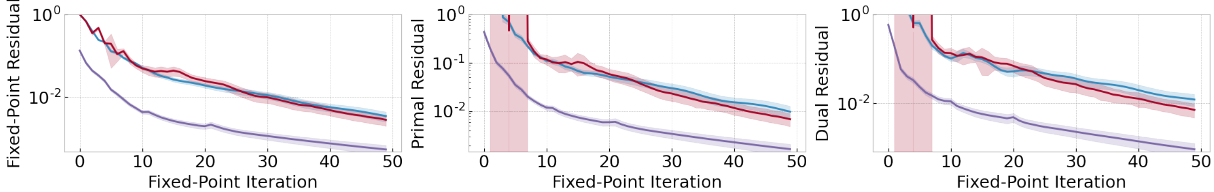
Additionally, we use *overparametrization* (Arora et al., 2018; Saunshi et al., 2020) to accelerate learning the models for the Kalman Filtering; this overparametrization increases the number of parameters in a given network without increasing the expressiveness. In our experiments, following that of Arora et al. (2018) on convolutional networks, we replace the matrix of each hidden layer of size $m \times m$ by two matrices of size $m \times m$, and the output layer of size $m \times k$ by two matrices of sizes $m \times k$ and $k \times k$ respectively.

Loss. As in Section 4, our loss is the *fixed-point residual norms* defined by $\mathcal{R}(x; \phi) \stackrel{\text{def}}{=} ||x - f(x; \phi)||_2$. With the intermittent access learning procedure, each iteration in the access set has a loss computed through fixed-point residuals. Our overall loss is the sum of all individual iteration losses; as before, we do not include the scaling factor τ in our loss (or in the models). To reduce the impact of the early iterates (whose residuals are much higher), we use the logarithm of the fixed-point norms.

Formally, we define our loss as follows. Let y_i denote the point iterate at iteration i. When the fixed-point residuals on y_{i+1} are computed for another r_i steps, the loss at iteration i is $\mathcal{L}_i = \sum_{j \leq r_i} \log \mathcal{R}(y_{i+j}; \phi)$. Further, for a pair of accesses $i_1, i_2 \in A$, if $r_i > i_2 - i_1$, then fixed-point residuals (and therefore the loss $\mathcal{L}_i$) include a model access for i_2 as well. We compute our total loss $\mathcal{L} = \sum_{i \in A} \mathcal{L}_i$ and differentiate through it.



(a) Lasso: $A = \{1, 2, 5, 10, 15\}, R = \{r_0 = 5; r_1, r_2, r_5 = 10; r_{10}, r_{15} = 20\}$



(b) Kalman filtering (uses overparametrization): $A = \{1, 2, 5, 10, 20\}, R = \{r_0, r_1, r_2 = 15; r_5, r_{10}, r_{20} = 30\}$

SCS SCS+AA SCS+Neural-MLP

Figure 2: Learned acceleration models in SCS+Neural-MLP improve on upon SCS/SCS+AA to reach a better solution with only a few intermittent model accesses.

5.3 Experiments

We show experimental results on the same two second-order cone problems of Section 4: Lasso and Robust Kalman Filtering. To distinguish the intermittent access acceleration model from the recurrent model, we will refer to it as *SCS+Neural-MLP*.

5.3.1 Experimental Setup

Our experimental setup here is identical to that of Section 4 to facilitate a comparison. The primary changes come from having to additionally set up the access set and residual intervals for the model. For SCS+Neural-MLP, we allow the access set to have 5 accesses to g_θ^{acc}, in addition to the access to g_θ^{init}. Our hyperparameter sweep now includes access set and residual intervals, which are key design elements that affect the accuracy and efficiency of the models –in general, we find that earlier accesses are more useful than later ones, and it is useful for the residual intervals to be at least long enough so that they go a few iterations past the next model access; for some applications, the residual intervals need to be much longer than others. Table 4 shows the range of values used in the hyperparameter sweep, and our results use the best choices that we found for each problem. All experiments are run with 3 seeds for repeatability, and each metric is aggregated over all test set instances and all runs.

5.3.2 Results

Figure 2 shows the fixed-point, primal and dual residuals for each problem, similar to Figure 1. Below, we discuss our results for each problem individually.

Lasso. Our models for g_θ^{init} and g_θ^{acc} both have 3 hidden layers, with 2560 and 5120 units respectively. In addition, as mentioned earlier, the accuracy of the learned model strongly depends on the choice of the access sets and the residual intervals. Through extensive hyperparameter search, we found the following access sets and residual intervals to work well for Lasso the access set is $A = \{1, 2, 5, 10, 15\}$, with the residual interval at iteration 0 $r_0 = 5$, at iterations 1, 2, 5 r_1, r_2, r_5 all as 10, and at iterations 10, 15 r_{10}, r_{15} as 20. For ease of notation, we denote this set of residual intervals as $R = \{r_0 = 5; r_1, r_2, r_5 = 10; r_{10}, r_{15} = 20\}$ in Figure 2.

Figure 2a shows the residuals of SCS+Neural-MLP; these are similar to the recurrent models in Section 4, but they require only 5 model accesses compared to the 50 required by the recurrent models. SCS+Neural-MLP is able to improve on the residuals of SCS by as many as 20 iterations (*i.e.*, by iteration 20, SCS+Neural-MLP

		Lasso (orig.)	Lasso: $q = 40$	Lasso: $q = 60$	dynamic Lasso (2-step)	dynamic Lasso (3-step)
	Total variables	22	22	22	53	83
	Constraints	44	64	84	105	165
	nnz in A	244	444	644	525	805
Cones	Zero	0	0	0	10	20
	Non-negative	20	20	20	40	60
	Second-order	[21, 3]	[41, 3]	[61, 3]	[41, 3, 11]	[61, 3, 21]

Table 2: Lasso variants for Section 6: problem sizes when in standard form

has improved over the residuals reached by SCS at iteration 40) and over SCS+AA by as many as 12 iterations (SCS+AA reaches the same residuals around iteration 32). SCS+Neural-MLP maintains all its improvement over SCS across most of the 50 iterations, but its improvement over SCS+AA starts to degrade past iteration 35 and disappears completely around iteration 45. Unlike Kalman filtering, overparametrization does not help improve the neural acceleration model for Lasso.

Kalman Filtering. For Kalman filtering, we use a 3-layer network for g_θ^{init} and g_θ^{acc}. We use 15000 units in the hidden layer for g_θ^{init}, and 7680 units in the hidden layer for g_θ^{acc}. We also use overparametrization in the hidden layer and the output layer, as described in Section 5.2. Our access set is $A = \{1, 2, 5, 10, 20\}$. We use a residual interval of 15 for the accesses at iterations 0, 1 and 2, and we use 30 for accesses at 5, 10, and 20. For ease of notation, we denote these as $R = \{r_0, r_1, r_2 = 15; r_5, r_{10}, r_{20} = 30\}$.

Our results are shown in Figure 2b. We note we are able to improve the convergence of all three residuals for SCS+Neural-MLP over SCS/SCS+AA by almost 5× fewer iterations: by iteration 10, SCS+Neural-MLP has already reached the same residuals that SCS and SCS+AA reach by iteration 50. This is unlike the recurrent models (shown in 1b in Section 4), where only the fixed-point residual has improved by iteration 10; the corresponding improvement in primal/dual residuals is very little. Further, unlike the recurrent models, the improvement over SCS and SCS+AA remains nearly as high at iteration 50 as it is at iteration 1. Indeed, the improvement far exceeds that of recurrent models in Section 4 – the primal/dual residuals for MLP models at iteration 10 are 10× already smaller than the recurrent models; at 50 iterations, all three residuals of SCS+Neural-MLP are at least 10× smaller than the recurrent models.

Thus, we see that the increased training efficiency achieved by our intermittent access framework and our modeling have enabled significant improvements over SCS/SCS+AA on Kalman filtering. In our experiments, overparametrization turns out to be crucial for the Kalman filtering results. The overparametrized model achieved the results shown in 30k-35k training iterations, while the regular model did not obtain similar results in 50k iterations.

5.3.3 Discussion

Our results show that SCS+Neural-MLP is able to accelerate the Kalman filtering problem distribution much better than the Lasso problem distribution. We note that using the same models, overparametrization, and hyperparameters as Kalman filtering for Lasso does not result in any noticeable improvement over the results in Figure 2. A natural question this raises is why this improvement is possible for Kalman filtering, but not Lasso.

To explore this question, we begin by identifying a few structural differences between these two problems:

- *Distributions of the A, b, c.* For Kalman Filtering, A is identical across all problem instances; only the b varies, and only in the entries corresponding to the zero cone. For Lasso, A, b and c are all different across the problem instances; however, c is identical subject to a normalization. In addition, b varies in the entries corresponding to second-order cone of size 101.

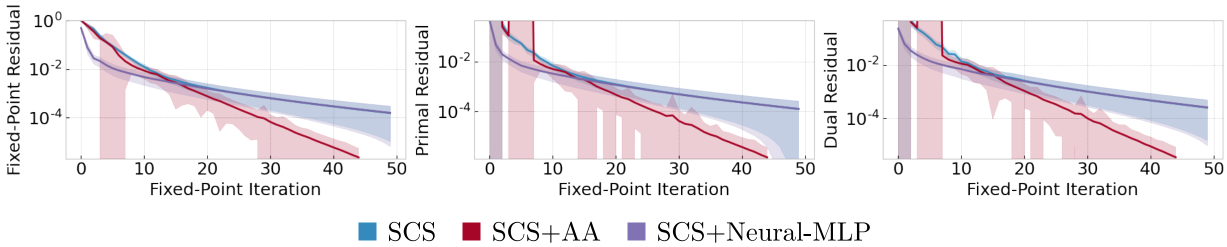


Figure 3: Original Lasso with $p = 10$, $q = 20$. Learned model uses $A = \{1, 2, 5, 10\}, r_i = 10$ for all i.

- *Cone structure.* The Kalman filtering problem has a zero cone of size 100 (from its equality constraints), while the Lasso problem has no zero cone. In addition, while both problems have second-order cones, the Robust Kalman Filtering has many (100) second-order cones of dimension 3, and one larger cone of dimension 102. The Lasso, instead, has two second-order cones, one of dimension 101, and one of size 3. This also induces a substantial difference in the sparsity and block-structure of the source matrices A of the two problems.

These differences lead to two major reasons why neural acceleration is more beneficial for Kalman filtering. First, the Kalman filtering problem is learning to solve for the same A with each of different instances; thus, this provides many more training instances and iterations for the same A, and reduces the amount of randomness in the problem. Second, the Kalman filtering problem has to fit a large zero cone; geometrically, the zero cone is a point, more restrictive than a second-order cone, and so all the algorithms will need to fit the solution exactly; this makes the problem harder for SCS and SCS+AA.

Based on these insights, in the next section, we design a series of modifications that convert a Lasso problem to a linear dynamical optimization problem like Kalman filtering, and show that these modifications improve the benefit of neural acceleration.

6 Variations between Optimization Problems

In this section, we explore some of the characteristics of second-order cone problem distributions that allow for the gain of significant improvements with neural acceleration. For this analysis, we begin with the original Lasso problem distribution, and examine whether, by applying modifications to either just the distribution or the optimization problem itself, the resulting problem distribution is easier for or benefits more from learned acceleration.

To simplify our analysis, we use a much smaller problem (than Section 5) from the Lasso distribution. We see that even these smaller problems are not easy for neural acceleration, even when using the same model as the larger sized Lasso problems. Each modification then changes the optimization problem and/or distribution such that it adds some properties of the Kalman filtering, and we train new neural acceleration models on the new distributions. This allows us to gain insight about the characteristics that make a problem distribution better for neural acceleration. Since these experiments are on smaller-sized problems, rather than the original Lasso and Kalman filtering problem distributions, we emphasize that these results are only indicative of the underlying properties. Nevertheless, our results suggest that linear dynamical systems may be a class of optimization problems that benefits from neural acceleration.

Our experimental setup for this section follows Section 5.3, with one difference: since the optimization problem distribution varies in each experiment, we will describe those in their respective sections. However, for ease of reference, Table 2 shows the problem sizes for the different Lasso variants when they are represented in standard form.

6.1 Optimization Problems of the Original Distribution

Our first experiment repeats the distribution of Section 5 on the smaller problem size, to provide a baseline for the remaining experiments. For these experiments, we use $p = 10, q = 20$, i.e. 10 variables and 20

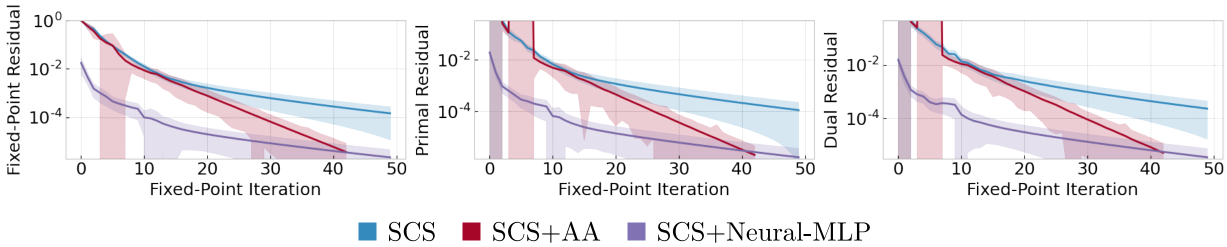


Figure 4: Reducing the randomness in the Lasso problem distribution by keeping the source matrix F fixed, leading to a fixed A matrix in standard form. Learned model uses $A = \{1, 2, 5, 10\}$, $r_i = 10$ for all i.

.

observations. We also observe that while each problem instance is generated with an underlying solution z^*, the addition of the noise w changes the optimization problem, and depending on the accuracy of the required residuals, z^* may no longer suffice as a solution.

Figure 3 shows the results of training SCS+Neural-MLP for this distribution. We see that there is only a very modest improvement over SCS, most of which is obtained around fixed-point iteration 5, and is fully lost after fixed-point iteration 10. We observe also that the SCS+Neural-MLP can predict to a obtained to a residual of 10^{-2}, and no improvement is obtained after that.

In Appendix C.1, we show that this trend does not change significantly for this distribution as the sparsity of the solution changes, i.e., as the number of non-zero values in z^* (termed the *density* ρ) changes from 0.1 to 0.3 and 0.5.

6.2 Reducing the Randomness in the Distribution

We now examine the effect of reducing the amount of randomness in the distribution. Recall that the original Lasso distribution has three independent components of randomly generated data: the source matrix F, solution z^*, noise w. (Note also that the quantity μ depends on both F and z^*, and affects the objective of the optimization problem in standard form.)

In our experiments below, we show the effect of keeping the source matrix F fixed (Section 6.2.1), and of reducing the noise in the distribution (Section 6.2.2). In Appendix C.2, we also include results that show that keeping the solution or the noise fixed does not affect the neural acceleration.

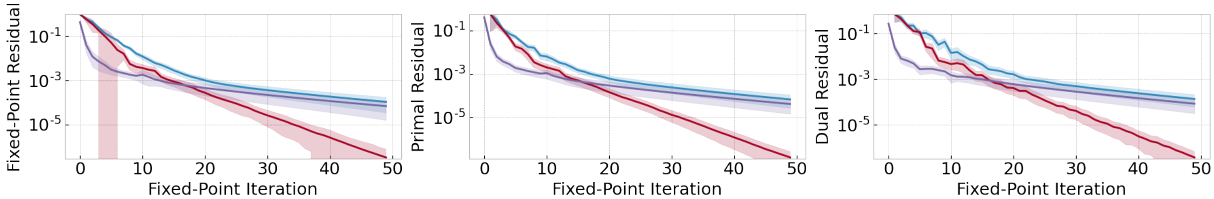
6.2.1 Fixed Source Matrix

In this experiment, we keep the source matrix F fixed throughout the problem distribution: i.e., we generate one source matrix F, and use it to generate all problem instances, drawing z^* and w at random as before. To ensure that we observe general trends, rather than dependencies on any single source matrix, we aggregate our results over problem distributions for 10 different source matrices.

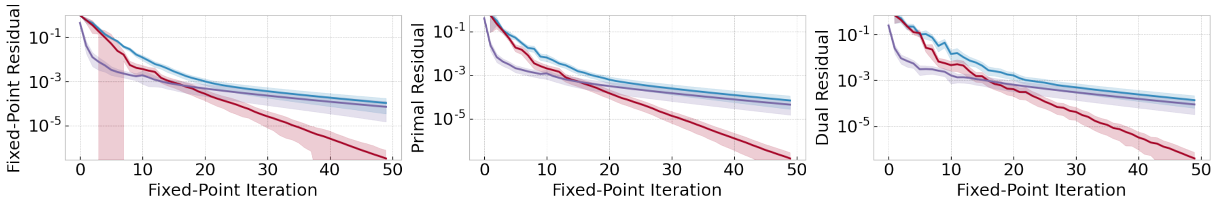
Figure 4 shows the results for SCS, SCS-AA, and SCS+Neural-MLP on this distribution. We note that the performance of SCS and SCS-AA does not change noticeably compared to Figure 3, but SCS+Neural-MLP has a dramatic improvement: now, SCS+Neural-MLP improves over SCS at all 50 fixed-point iterations, typically by over an order of magnitude, and it gets primal/dual residuals as low as 10^{-5} in around 35-40 iterations. SCS+Neural-MLP even improves substantially over SCS-AA as well over the first 30 fixed-point iterations, and for the first 10 iterations, it maintains an improvement of 2 orders of magnitude over SCS-AA as well.

6.2.2 Reduced Noise

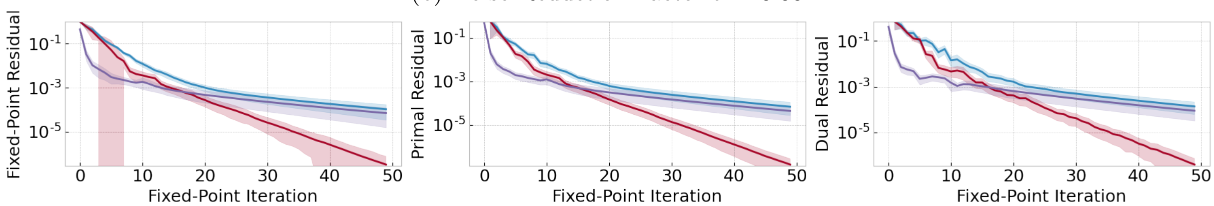
Our next experiment shows that just reducing the noise in the original distribution (while allowing the source matrix F to change) is not sufficient to learn to optimize well on this Lasso problem distribution.



(a) Noise Reduction Factor $\sigma = 0.01$



(b) Noise Reduction Factor $\sigma = 0.001$



(c) Noise Reduction Factor $\sigma = 0.0001$

SCS SCS+AA SCS+Neural-MLP

Figure 5: Reduced noise in Lasso distribution. Learned models use $A = \{1, 2, 5, 10\}, r_i = 10$ for all i.

For this experiment, we modify the problem distribution as follows: we generate F, z^* and w as earlier, but we define $g = Fz^* + \sigma w$, where σ is a multiplicative factor that reduces the impact of the noise on the final solution. In our experiment, we use $\sigma = 10^{-2}, 10^{-3}, 10^{-4}$. Note that in our original distribution $\sigma = 0.1$.

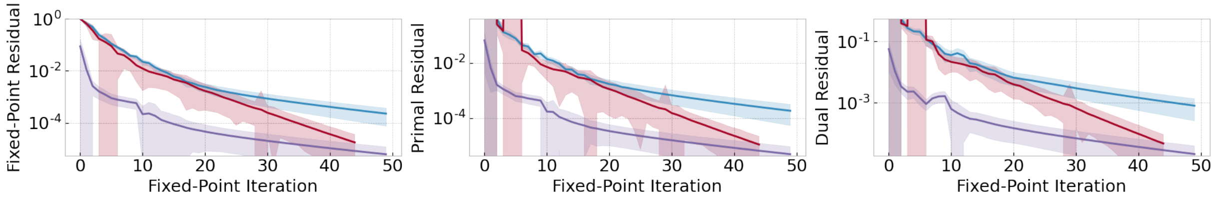
Fig. 5 shows the results. We see that for all three σ values, there is an improvement over the original distribution, as the SCS+Neural-MLP is now able to obtain improvements over SCS to at least 10^{-3}, which is achieved around fixed-point iteration 10. We note also that there does not appear to be a noticeable difference between SCS+Neural-MLP's performance on the different σ values: $\sigma = 0.01$ already obtains the maximum improvement if only the noise multiplication factor is reduced.

These results suggest that even the F and z^* in the Lasso distribution have too much variability for this particular model size/design to learn to optimize well.

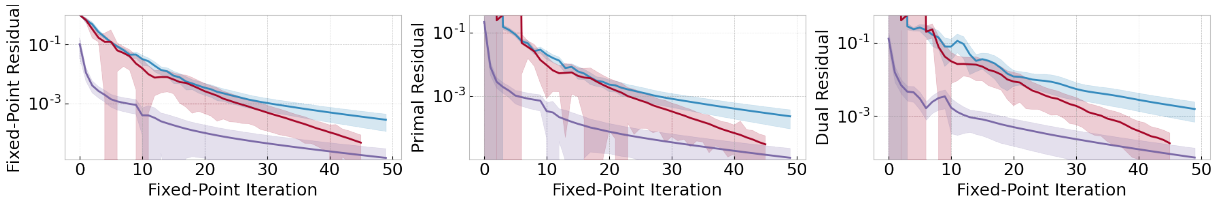
6.3 From Lasso to a Linear Dynamical System

Our last set of experiments explore the ease of neural acceleration when the Lasso problem we have studied above is converted to a linear dynamical system, like the Kalman filtering problem. A linear dynamical system evolves (noisily) over time, so we will have many more observations on the same matrix. However, more observations typically reduce the space of valid good solutions (since the noise in our problem distributions are generated independently), which might make it easier for both SCS and SCS+Neural to find the optimization solution z.

To isolate the impact of the dynamical system from the increased number of observations, we first increase the number of observations with the variables remaining static, in Section 6.3.1. Then, in Section 6.3.2, we modify our problem into a linear dynamical system by allowing the solution to move gradually in a defined manner as well over time steps, and obtain one set of observations at each time step.



(a) $q = 40$, density $\rho = 0.1$



(b) $q = 60$, density $\rho = 0.1$

SCS SCS+AA SCS+Neural-MLP

Figure 6: Increased observations for fixed source Lasso distributions. Learned models use $A = \{1, 2, 5, 10\}, r_i = 10$ for all i.

6.3.1 Static Solution

In this experiment, we use a Lasso problem distribution very similar to Section 6.2.1, but with increased observations. The distribution of Section 6.2.1 used fixed source matrices of size $q \times p$ where $q = 20$ and $p = 10$. Here we increase the number of observations to $q = 40$ and $q = 60$ respectively.

Figure 6 shows the results for these problem distributions. We observe that while the residuals for all three algorithms (SCS, SCS-AA and SCS+Neural-MLP) are larger than in Figure 4, the improvement of the learned models of SCS/SCS-AA is slightly increased, exceeding 2 orders of magnitude throughout. The higher residuals can likely because having more observations q makes it harder for SCS/SCS-AA to fit the noise as well in the optimization solution z. The results for $\rho = 0.3$ and $\rho = 0.5$ are similar, and hence we do not include them here.

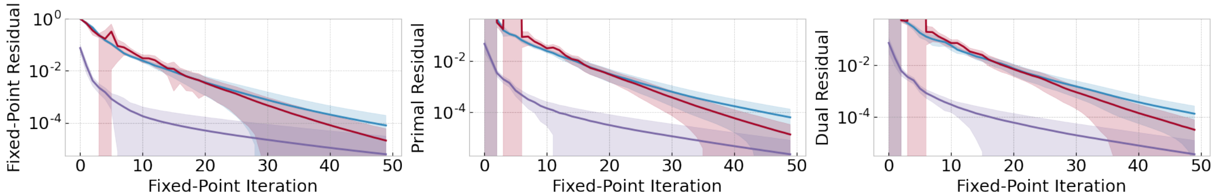
6.3.2 Dynamic Solution

In our final experiment, we modify the Lasso into a linear dynamical system, simulating the (defined) movement of the Lasso solution over multiple time steps, and show its impact on the benefits of neural acceleration.

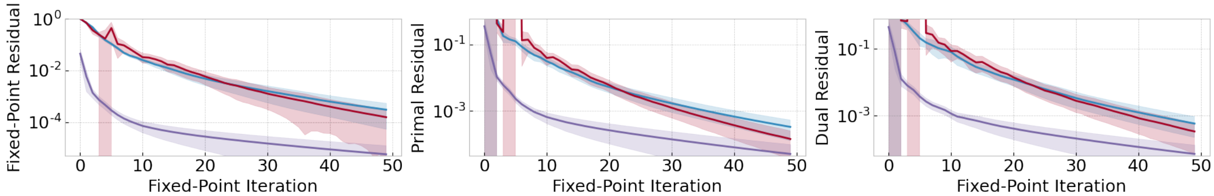
Problem Generation. Concretely, we modify our problem generation process as follows:

- We compute a source matrix F, and an initial solution z_1^* as before, where z_1^* has the required density ρ. We compute $w_1, w_2 \dots$ as the noise, as independent vectors for each required step.
- For each step, we move the solution z^* forward by a defined step size δ, e.g., $z_{i+1}^* = z_i^* + \delta$. Only the non-zero entries of z_i^* are changed in order to maintain sparsity.
- We then compute $g_i = F z_i^* + w_i$ as the output solution for each required step. As is typical in optimization problems derived from linear dynamical systems, we penalize all the noise by a reduced factor of σ.

With this process, we define our optimization problem as follows. Let H denote the source matrix created by repeating F for time t number of times in a block structure; let g denote the vector of observations $[g_1; g_2 \dots g_t]$, and let z denote the vector of all the solutions $[z_1; z_2 \dots z_t]$; and let v denote the vector of all



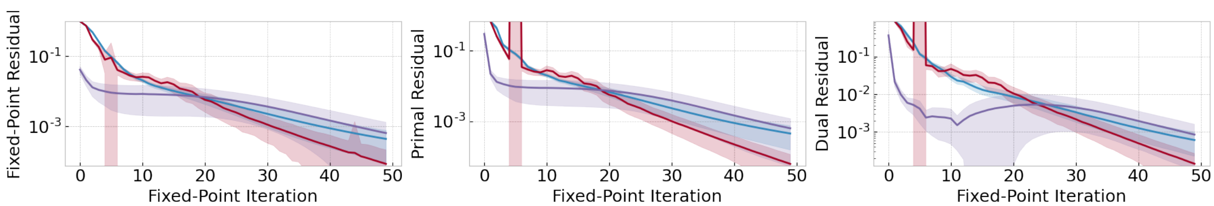
(a) 2 steps forward, density $\rho = 0.1$



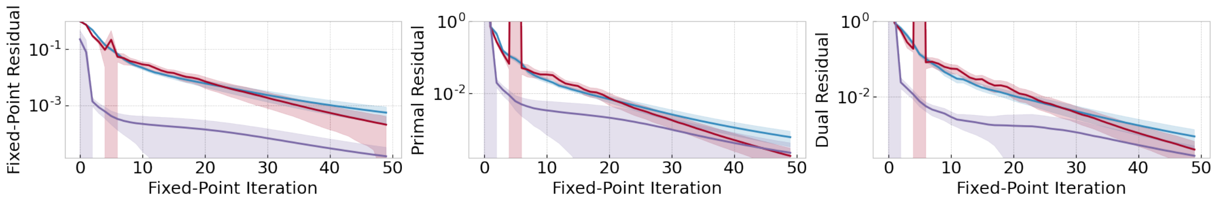
(b) 3 steps forward, density $\rho = 0.1$

SCS SCS+AA SCS+Neural-MLP

Figure 7: A simple linear dynamical system that simulates the Lasso solution moving in time. The problem distribution has a fixed source matrix F, but varying z^* per instance, and varying w per instance and per time step. Learned models use $A = \{1, 2, 5, 10\}, r_i = 10$ for all i.



(a) 2 steps forward, density $\rho = 0.5$



(b) 3 steps forward, density $\rho = 0.5$

SCS SCS+AA SCS+Neural-MLP

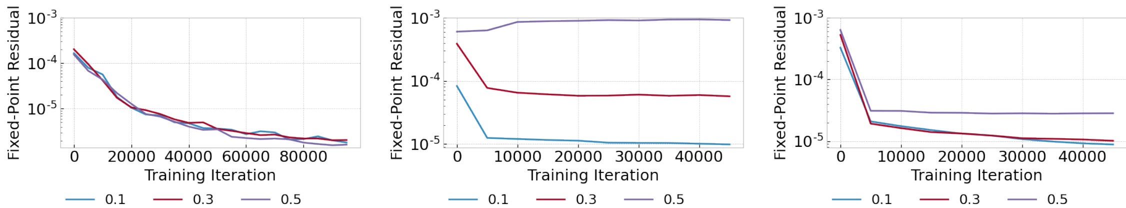
Figure 8: Linear dynamical system simulating the Lasso forward in time with density $\rho = 0.5$. Learned models use $A = \{1, 2, 5, 10\}, r_i = 10$ for all i.

noise $[v_1; v_2 \dots v_n]$.

$$\begin{aligned} \text{minimize}\, & ||Hz - g||_2^2 + \lambda ||z||_1^2 + ||v||_2 \\ \text{s. t. } & z_{i+1} = z_i + \delta + \beta v_i \end{aligned}$$

In our experiments, we set $\delta = 0.1$ and $\beta = 0.005$. All other parameters remain as before from the Lasso problem earlier.

Results. Figure 7 shows the results on this problem distribution for $\rho = 0.1$ (the results for $\rho = 0.3$ are similar); we have simulated the linear dynamical system described above 2 steps and 3 steps forward to obtain the problem distributions. We first note that this problem distribution is, overall, harder for SCS-AA to improve over SCS – note that the residuals obtained by SCS-AA are very close to that of SCS. However,



(a) Lasso: fixed source matrix F (b) Dynamic Lasso: 2 steps forward (c) Dynamic Lasso: 3 steps forward

Figure 9: Residuals as a function of training iterations. Learned models use $A = \{1, 2, 5, 10\}, r_i = 10$ for all i.

SCS+Neural-MLP is able to substantially improve on SCS and SCS-AA, by well over 2 orders of magnitude. Indeed, the relative improvement over SCS exceeds even that of the fixed source matrix Lasso distributions in 6.2.1.

Next, Figure 8 shows the results when $\rho = 0.5$. Note that here, the results are more mixed: when there are only 2 steps, the model learns very poorly, but with 3 steps, the model is able to learn quite well, although not as well as in Figure 7, with $\rho = 0.1$.

Finally, we explore how quickly SCS+Neural-MLP learns a good model for the different variations in Figure 9. We see that for the dynamic Lasso (with 3 steps), SCS+Neural-MLP learns a good model very quickly, reaching close to its best residual in as little as 5000 training iterations; this is similar to what we observed in the Robust Kalman Filtering models. In contrast, the model learnt by SCS+Neural-MLP for fixed source matrix distributions learns slowly and gradually over (at least) 50000 training iterations. Taken together, these results suggest that linear dynamical systems may be a class of optimization problems that benefit from neural acceleration.

7 Conclusion

Continuous fixed-point problems are a computational primitive in numerical computing, optimization, machine learning, and the natural and social sciences, and have recently been incorporated into deep learning models as optimization layers. Acceleration of fixed-point computations has traditionally been explored in optimization research without the use of learning. In this work, we introduce neural fixed-point acceleration, a framework to automatically learn to accelerate fixed-point problems that are drawn from a distribution; a key question motivating our work is to better understand the characteristics that make neural acceleration more beneficial for some problems than others. We apply the framework to solve second-order cone programs with the Splitting Conic Solver (SCS), and evaluate on distributions of Lasso problems and Kalman filtering problems. Our main results show that we are able to get a 10× performance improvement in accuracy on the Kalman filtering distribution, while those on Lasso are much more modest. We then isolate a few factors that make neural acceleration much more useful on the Kalman filtering distribution than on the Lasso distribution; we apply a number of problem and distribution modifications on a scaled-down version of the Lasso problem, adding in properties that make it structurally closer to Kalman filtering, and show when the problem benefits from neural acceleration. Our results suggest that linear dynamical systems may be a class of optimization problems that benefit from neural acceleration.

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*, pp. 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*, pp. 70–79. PMLR, 2017.

Brandon Amos. Tutorial on amortized optimization for learning to optimize over continuous domains, 2022. URL https://arxiv.org/abs/2202.00665.

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*, pp. 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*, pp. 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*, pp. 3981–3989, 2016.

Sanjeev Arora, Nadav Cohen, and Elad Hazan. On the optimization of deep networks: Implicit acceleration by overparameterization. In *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pp. 244–253. PMLR, 10–15 Jul 2018.

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.

Shaojie Bai, Vladlen Koltun, and J Zico Kolter. Neural deep equilibrium solvers. In *International Conference on Learning Representations*, 2022. URL https://openreview.net/forum?id=B0oHOwT5ENL.

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.

Nicola Bastianello, Andrea Simonetto, and Emiliano Dall'Anese. Opreg-boost: Learning to accelerate online algorithms with operator regression. *arXiv preprint arXiv:2105.13271*, 2021.

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

Stephen Boyd and Lieven Vandenberghe. *Convex optimization*. Cambridge university press, 2004.

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.

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, 2022. Accessed: 2022-07-14.

Justin Domke. Generic methods for optimization-based modeling. In *AISTATS*, volume 22, pp. 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*, pp. 1126–1135. PMLR, 2017.

John R. Giles. *Introduction to the Analysis of Metric Spaces*. Cambridge University Press, 1987.

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*, pp. 399–406, 2010.

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

Jeffrey Ichnowski, Paras Jain, Bartolomeo Stellato, Goran Banjac, Michael Luo, Francesco Borrelli, Joseph E. Gonzalez, Ion Stoica, and Ken Goldberg. Accelerating quadratic optimization with reinforcement learning, 2021.

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*, pp. 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.

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.

Arkadi Nemirovski. Advances in convex optimization: Conic programming. In *In Proceedings of International Congress of Mathematicians*, pp. 413–444, 2007.

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.

Nikunj Saunshi, Yi Zhang, Mikhail Khodak, and Sanjeev Arora. A sample complexity separation between non-convex and convex meta-learning. In *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pp. 8512–8521. PMLR, 13–18 Jul 2020.

R.S. Sutton and A. G. Barto. *Reinforcement learning: An introduction.* MIT Press, 2018.

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*, pp. 3751–3760. PMLR, 2017.

Yuhuai Wu, Mengye Ren, Renjie Liao, and Roger Grosse. Understanding short-horizon bias in stochastic meta-optimization. In *International Conference on Learning Representations*, 2018.

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.

A Dynamics matrices for Kalman filtering

For ease of reference, we include here the full dynamics matrices from Diamond & Boyd (2022) used in the robust Kalman filtering problem.

$$F = \begin{bmatrix} 1 & 0 & (1-\frac{\gamma}{2}\Delta t)\Delta t & 0 \\ 0 & 1 & 0 & (1-\frac{\gamma}{2}\Delta t)\Delta t \\ 0 & 0 & 1-\gamma\Delta t & 0 \\ 0 & 0 & 0 & 1-\gamma\Delta t \end{bmatrix} \qquad G = \begin{bmatrix} \frac{1}{2}\Delta t^2 & 0 \\ 0 & \frac{1}{2}\Delta t^2 \\ \Delta t & 0 \\ 0 & \Delta t \end{bmatrix} \qquad H = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

B Hyperparameters used for training

Table 3 and Table 4 show the range of parameters used in the hyperparameter sweep for SCS+Neural (with recurrent models) and SCS+Neural-MLP (with MLP models) respectively.

C Additional Variations

In this appendix, we include additional variations on the Lasso problems that provide insight into how the modifications affect whether an optimization problem benefits from neural acceleration.

C.1 Changing Solution Sparsity in Original Distribution

In this experiment, we show that the trends described in Section 6.1 essentially hold when the sparsity of the solution changes. Recall that, in our distribution above, only 10% of the variables (1 variable with $p = 10$) are set to non-zero values in z^*. Here we show the effect of reducing the sparsity of the solution. We define the fraction of non-zero variables in the solution z^* as the *density* of the solution. In the experiments that folow, we see how the previous trends change as the density ρ increases from 0.1 to 0.3 and 0.5.

Table 3: Parameters used for hyperparameter sweep of SCS+Neural with Recurrent Models

Adam	
learning rate	10^{-4}, 10^{-2}
β_1	0.1, 0.5, 0.7, 0.9
β_2	0.1, 0.5, 0.7, 0.9, 0.99, 0.999
cosine learning rate decay	True, False
Misc	
max gradient for clipping	10.0, 100.0
batch size	16, 32, 64, 128 [Lasso] 5, 10, 25, 50 [Kalman filter]

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 . . . 4]
Encoder:	
- hidden units	128, 256, 512, 1024
- activation function	relu, tanh, elu
- depth	[0 . . . 4]
Decoder:	
- hidden units	128, 256, 512, 1024
- activation function	relu, tanh, elu
- depth	[0 . . . 4]
- weight scaling	[2.0, 128.0]
Recurrent Cell:	
- model	LSTM, GRU
- hidden units	128, 256, 512, 1024
- depth	[1 . . . 4]

Table 4: Parameters used for hyperparameter sweep of SCS+Neural-MLP

Adam	
learning rate	[10^{-4}, 10^{-2}]
β_1	0.9
β_2	0.999
cosine learning rate decay	False
Misc	
max gradient for clipping	100.0
batch size	32, 64, 128 [Lasso] 25, 50 [Kalman filter]

Neural Model	
- use initial hidden state	True
- use initial iterate	True
- use overparametrization	True, False
MLP:	
- hidden units	1024, 1280, 2560, 5120, 7680, 10240, 15000
- activation function	relu
- depth	[2 . . . 4]
- number of overparameterization layers	[1 . . . 4]
Access sets:	Various subsets of [1, 2, 5, 8, 10, 15, 20, 30, 40] Every x iterations, for x in [3, 5, 10]
Residual Intervals:	[3, 5, 10, 15, 20, 30] Every x times the gap between consecutive elements in the access set, for x = [1, 1.5, 2, 3]

Our experiment shows that the results from Section 6.1 change only slightly when the density of z^* is increased to 0.3 and 0.5. Figure 10 shows the results of learning on these distributions for SCS, SCS-AA and SCS+Neural-MLP. We note that the increased density allows for slightly increased performance of SCS+Neural-MLP, i.e., at $\rho = 0.5$, the residuals of SCS-Neural now reach 5e-3 before they no longer improve over SCS.

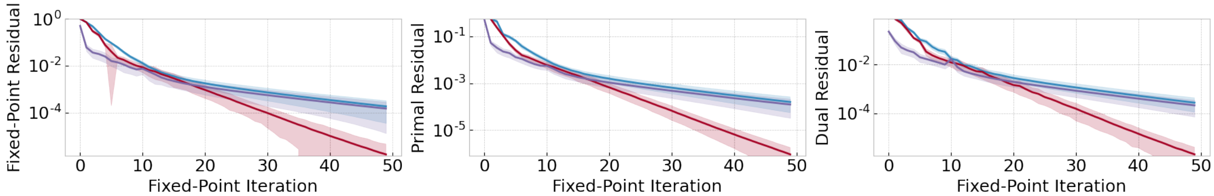
C.2 Fixed Solution/Noise Matrix

In this section, we include additional experiments from Section 6.2, showing that keeping the noise w fixed or the solution z^* does not result in a distribution where neural acceleration is beneficial.

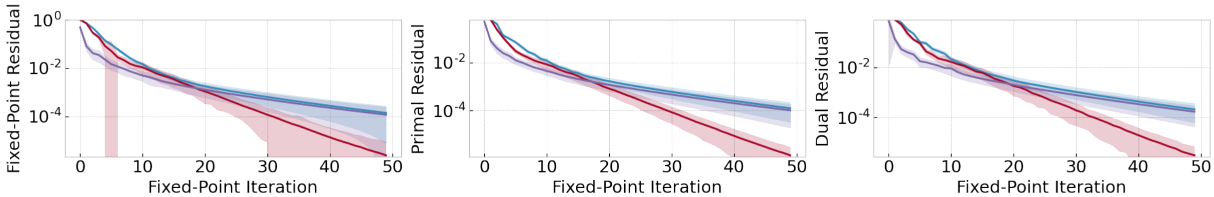
C.2.1 Fixed Noise Matrix

Our next experiment examines how well the model learns when the noise is kept fixed. In this experiment, we generate one fixed w, and then use it to generate all problem instances similar to the original distribution, i.e., we draw F, z^* from the same distributions as the original, and then use our fixed w to generate $g = Fz^* + w$. μ is generated as before.

Figure 11a shows the results for SCS, SCS-AA and SCS+Neural-MLP for this distribution. We note that the improvement of SCS+Neural-MLP over SCS here is again modest, only to a fixed-point residuals around 10^{-3}; indeed, the improvement is a little less than those in Figure 5. This is consistent with our earlier finding – a problem distribution where the noise is fixed is no better than a problem distribution where the noise is much smaller than the required residuals.



(a) Density $\rho = 0.3$



(b) Density $\rho = 0.5$

SCS SCS+AA SCS+Neural-MLP

Figure 10: Original Lasso distribution with increased density ρ (Appendix C.1). Learned models use $A = \{1, 2, 5, 10\}, r_i = 10$ for all i.

C.2.2 Fixed Solution

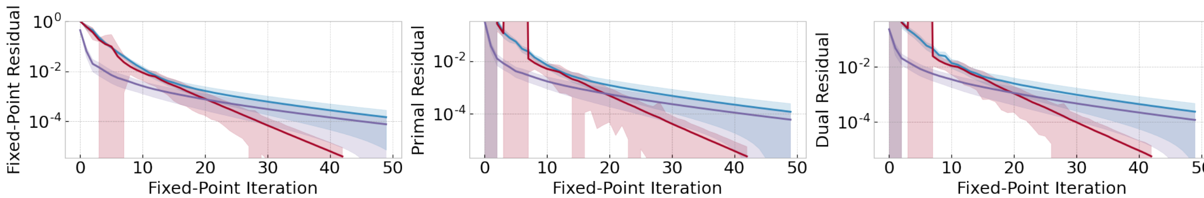
Our next experiment examines how well the model learns when the original solution z^* is kept fixed. In this experiment, we generate one fixed z^*, and use it to generate all the problem instances as we did in our original distribution, i.e., we draw F, w from the same distribution as we did originally, and then use our fixed z^* to generate $g = Fz^* + w$. Again, μ is generated as before. Again, we generate datasets with 10 different solutions, and learn models on each of them.

Figure 11b shows the results for SCS, SCS-AA, SCS+Neural-MLP aggregated over the 10 different datasets. We see that again, SCS and SCS-AA are similar to the previous distributions, but SCS-Neural is now no longer able to learn much. Indeed, the performance of SCS-Neural is now similar to that in the original Lasso distribution (Figure 3), where the most substantial improvement over SCS comes at fixed-point iteration 5 with a residual of 10^{-2}.

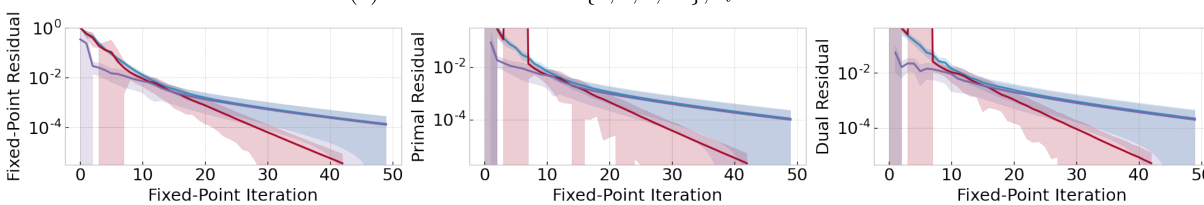
D Importance of τ Normalization

In this appendix, we show the importance of normalizing the fixed-point residual norms for the loss by the τ conditioning factor in SCS. Figure 12 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 Section 4, this happens when SCS+Neural achieves a low loss by simply learning a low τ.

We verify this empirically examining how τ changes over the fixed-point iterations. Figure 13 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.



(a) Fixed Noise: $A = \{1, 2, 5, 10\}, r_i = 10$ for all i

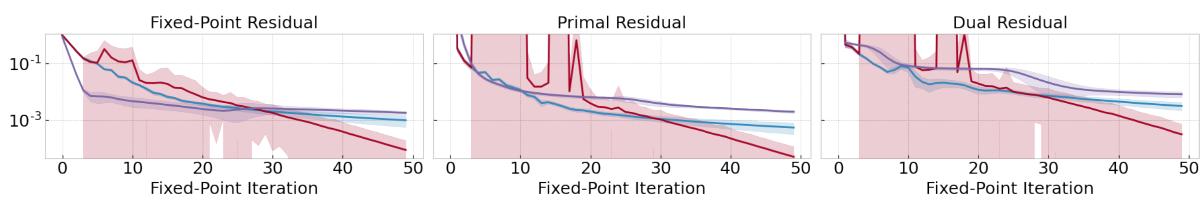


(b) Fixed Solution: $A = \{1, 2, 5, 10\}, r_i = 10$ for all i

SCS SCS+AA SCS+Neural-MLP

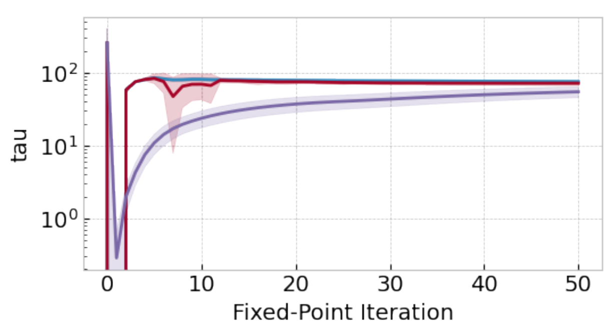
Figure 11: Reducing the randomness in the Lasso problem distribution by keeping the noise or the solution fixed (Appendix C.2)

.



SCS SCS+AA SCS+Neural (No τ normalization)

Figure 12: Lasso without τ normalization: a failure mode of neural acceleration (that SCS+Neural overcomes with design), see Appendix D.



SCS SCS+AA SCS+Neural (No τ normalization)

Figure 13: 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 and does not solve the optimization problem well (Appendix D).