Journal of Machine Learning Research 24 (2023) 1-26 Submitted 9/22; Published 10/23

Sensitivity-Free Gradient Descent Algorithms

Ion Matei ION.MATEI@SRI.COM
Intelligent Systems Laboratory
PARC, part of SRI International
Palo Alto, CA 94304, USA

Maksym Zhenirovskyy MAKSYM.ZHENIROVSKYY@SRI.COM
Intelligent Systems Laboratory
PARC, part of SRI International
Palo Alto, CA 94304, USA

Johan de Kleer JOHAN.DEKLEER@SRI.COM
Intelligent Systems Laboratory
PARC, part of SRI International
Palo Alto, CA 94304, USA

John Maxwell JOHN.MAXWELL@SRI.COM
Intelligent Systems Laboratory
PARC, part of SRI International
Palo Alto, CA 94304, USA

Editor: Sathiya Keerthi

Abstract

We introduce two block coordinate descent algorithms for solving optimization problems with ordinary differential equations (ODEs) as dynamical constraints. In contrast to prior algorithms, ours do not need to implement sensitivity analysis methods to evaluate loss function gradients. They result from the reformulation of the original problem as an equivalent optimization problem with equality constraints. In our first algorithm we avoid explicitly solving the ODE by integrating the ODE solver as a sequence of implicit constraints. In our second algorithm, we add an ODE solver to reset the estimate of the ODE solution, but no sensitivity analysis method is needed. We test the proposed algorithms on the problem of learning the parameters of the Cucker-Smale model. The algorithms are compared with gradient descent algorithms based on ODE solvers endowed with sensitivity analysis capabilities. We show that the proposed algorithms are at least 4x faster when implemented in Pytorch, and at least 16x faster when implemented in Jax. For large versions of the Cucker-Smale model, the Jax implementation is thousands of times faster. Our algorithms generate more accurate results both on training and test data. In addition, we show how the proposed algorithms scale with the number of optimization variables, and how they can be applied to learning black-box models of dynamical systems. Moreover, we demonstrate how our approach can be combined with approaches based on sensitivity analysis enabled ODE solvers to reduce the training time.

Keywords: gradient descent, sensitivity analysis, neural-ODEs, deep learning

1. Introduction

Various engineering applications in system design, control, or diagnosis are formulated as optimization problems with dynamical constraints. The most common mathematical models to describe system dynamics include ordinary differential equations (ODEs), differential algebraic equations (DAEs), or partial differential equations (PDEs). Such models rarely have closed-form solutions, requiring numerical approximation of their solutions via solvers. Computing gradients of loss functions requires using sensitivity analysis techniques to determine the changes in the model dynamics as the optimization parameters are varied. Dickinson and Gelinas (1976), and Cao et al. (2002) introduced the direct and adjoint sensitivity analysis methods, respectively. In direct methods, the sensitivity of the solution of an ODE $\dot{x} = f(x;\theta)$, where x and θ are the state vector and the model parameters, respectively, is explicitly computed by solving an associated ODE defined in terms of the Jacobian of the map $f(x;\theta)$. The direct method works well when the number of optimization parameters θ is small, but does poorly as the size of θ increases. In contrast, using the adjoint method there is no need to explicitly compute the sensitivity of the state x with respect to θ. The gradient of the loss function is computed by solving the adjoint ODE that scales linearly with the size of the state vector.

Recently, deep learning platforms such as Pytorch (Paszke et al. (2017)) or Jax (Bradbury et al. (2018)) were endowed with capability to include ODEs in their models. Computing gradients of loss functions over ODE solutions requires implementing sensitivity analysis methods (direct or adjoint) in the computational structure of these platforms. Regardless of the approach, as the complexity of these dynamical models grows so does the computational cost to solve them. Moreover, the time required to solve ODEs depends on the type of solver used (e.g., fixed/adaptive step, explicit/implicit) and on the numerical stability of the ODE. The latter is determined by the parameters explored during the optimization process and, at times, can destabilize the main ODE, or the ODE needed to be solved in the direct or adjoint methods. In this paper we introduce algorithms that remove the need to solve ODEs induced by direct or adjoint sensitivity analysis methods. Consequently, our algorithms are faster, since at most we have to solve only the main ODE. We are particularly interested in how the dimension of the state vector affects the computationally complexity of the optimization process. The reason for such an interest is that in applications such as model-based diagnosis we often use single or double fault assumptions that require estimating a small number of parameters compared the state dimension.

We first formulate the optimization problem with dynamical constraints. Let $x \in \mathbb{R}^n$ denote a state vector that satisfies an ODE $\dot{x} = f(x;\theta)$, where $\theta \in \mathbb{R}^p$ is a set of parameters of the ODE that also serve as optimization parameters for the learning problem. Let $y = h(x) \in \mathbb{R}^m$ be a vector representing indirect observations of the state vector x. We denote by $\hat{\boldsymbol{x}}(\theta) = [\hat{x}_1(\theta), \hat{x}_2(\theta), \ldots, \hat{x}_N(\theta)]$ and by $\hat{\boldsymbol{y}} = H(\hat{\boldsymbol{x}}) = [\hat{y}_1, \hat{y}_2, \ldots, \hat{y}_N]$ the solution of the ODE and its corresponding outputs, respectively, at time samples $\mathcal{T} = \{t_i\}_{i=0}^N$, with $h = t_{i+1} - t_i$. Let $L(\hat{\boldsymbol{y}}(\hat{\boldsymbol{x}}(\theta))) = (L \circ \hat{\boldsymbol{y}} \circ \boldsymbol{x})(\theta) = \bar{L}(\theta)$ be a scalar loss function (e.g., mean square error). Then an optimization problem with dynamical constraints can be expressed as

$$\min_{\theta} \bar{L}(\theta), \tag{1}$$

where $\hat{\boldsymbol{x}}(\theta) = \texttt{ODESolver}(\theta; x_0)$, x_0 is the initial condition of the ODE, and `ODESolver` generates the solution of the ODE at time instants $\mathcal{T}$. A first order gradient descent algorithm to solve (1) is given by

$$\theta_{k+1} = \theta_k - \alpha \nabla_\theta \bar{L}(\theta_k), \tag{2}$$

where $\nabla_\theta \bar{L}(\theta)$ is the gradient of the loss function with respect to the vector of parameters θ. The loss function gradient can be explicitly written as:

$$\nabla_\theta \bar{L}(\theta_k) = \left[\frac{\partial \hat{\boldsymbol{x}}}{\partial \theta}(\theta_k)\right]^T \left[\frac{\partial \hat{\boldsymbol{y}}}{\partial \hat{\boldsymbol{x}}}(\hat{\boldsymbol{x}}_k)\right]^T \nabla_{\hat{\boldsymbol{y}}} L(\hat{\boldsymbol{y}}_k),$$

where $\frac{\partial \hat{\boldsymbol{x}}}{\partial \theta}$ and $\frac{\partial \hat{\boldsymbol{y}}}{\partial \hat{\boldsymbol{x}}}$ are the Jacobian of the states with respect to θ, and of the outputs with respect to the states, respectively, and where $\hat{\boldsymbol{x}}_k = \hat{\boldsymbol{x}}(\theta_k)$ and $\hat{\boldsymbol{y}}_k = H(\hat{\boldsymbol{x}}_k)$. We summarize the iteration (2) in Algorithm 1.

Algorithm 1 Gradient Descent with Sensitivities-Enabled ODE Solver.

Require: α: Stepsize
Require: x_0: Initial state vector
Require: θ_0: Initial parameter vector
  $k \leftarrow 0$
  while θ_k not converge **do**
    $k \leftarrow k + 1$
    $\theta_k \leftarrow \theta_{k-1} - \alpha \nabla_\theta \bar{L}(\theta_{k-1})$
  end while
  return θ_k

In direct sensitivity analysis methods, key to the implementation of the gradient descent algorithm is the evaluation of the Jacobian of the state vector with respect to θ. This quantity reflects the sensitivity of the state with respect to the vector of parameters. To avoid using numerical approximations when evaluating $\frac{\partial \boldsymbol{x}}{\partial \theta}$, ODE solvers were augmented with the capability to compute sensitivities of the solution $x(t)$ with respect to θ. For example, the SUNDIALS software family, introduced by Gardner et al. (2022); Hindmarsh et al. (2005), with DAE solvers such as CVODES and IDAS, include both direct and adjoint-based approaches to compute sensitivities. When improving a model to account for behavior that is not captured in an initial model, as discussed in Psichogios and Ungar (1992); Meleshkova et al. (2021), one approach is to hybridize the ODE by augmenting it with new representations such as neural networks (NNs). Training the parameters of the NN would require the implementation of the learning problem on a deep learning (DL) platform such as Pytorch or Jax. Currently, the algorithms in the SUNDIALS library are not directly integrated with DL platforms. Such platforms are of interest, since the optimization algorithm can take advantage of the automatic differentiation (AD) feature when computing gradients of loss functions. Recently, DL platforms were endowed with capabilities that include ODE-based layers. For example, Chen et al. (2018) used the adjoint method to compute the gradient of the loss function, by extending the original ODE with an additional ODE representing the adjoint variable dynamics. Similar efforts to extend Jax with ODE simulation capabilities

were made by Hessel et al. (2020). Regardless of the direct or adjoint approach to account for state sensitivities, as the size of the state vector increases, the evaluation of the loss function gradient will be more costly: at least linear in the state dimension, even when discounting the sometimes unpredictable numerical stability of the ODE solvers.

We introduce two block coordinate descent algorithms (Bertsekas (1999)) that do not require sensitivity analysis methods to compute loss function gradients. The proposed algorithms are based on implicit constraints derived from the ODE and can be easily extended to dynamical models represented as DAEs. More importantly, they support batch executions over time samples that can be efficiently executed on GPUs. For a vector $\boldsymbol{x} = \boldsymbol{x}_{0:N} = [x_0; x_1; x_2 \ldots; x_N] \in \mathbb{R}^{n(N+1)}$, $\boldsymbol{x}_{1:N}$ is the vector $\boldsymbol{x}$ without x_0, and $\boldsymbol{x}_{0:N-1}$ is the vector $\boldsymbol{x}$ without the last entry x_N. For a function $f(x)$, $f(\boldsymbol{x}_{0:N}) = [f(x_0); f(x_1) \ldots; f(x_N)]$. We introduce the residual function $r(\boldsymbol{x}, \theta) : \mathbb{R}^{n(N+1)+p} \to \mathbb{R}^{nN}$ derived from the application of *direct collocation* methods. The residual function measures how close a sequence of state vectors $\boldsymbol{x}$ is to a solution of the ODE for which the residual function is defined. We give an example of such a residual function in Section 2. In addition to the residual function, we define the loss function $F(\boldsymbol{x}, \theta) = \tilde{L}(\boldsymbol{x}) + \frac{1}{2}\|r(\boldsymbol{x}, \theta)\|^2$, where $\tilde{L}(\boldsymbol{x}) = (L \circ \boldsymbol{y})(\boldsymbol{x})$.

Algorithm 2 Block coordinate gradient descent with residual functions based on implicit constraints.

Require: α_x, α_θ: Stepsizes
Require: $r(\boldsymbol{x}, \theta)$: Residual function as implicit dynamical constraints
Require: x_0: Initial state vector
Require: θ_0: Initial parameter vector
  $\boldsymbol{x}_0 \leftarrow$ `ODESolver`$(\theta_0; x_0)$
  $k \leftarrow 0$
  while θ_k, $\boldsymbol{x}_k$ not converge **do**
    $k \leftarrow k + 1$
    $\boldsymbol{x}_k \leftarrow \boldsymbol{x}_{k-1} - \alpha_x \nabla_{\boldsymbol{x}} F(\boldsymbol{x}_{k-1}, \theta_{k-1})$
    $\theta_k \leftarrow \theta_{k-1} - \alpha_\theta \nabla_\theta F(\boldsymbol{x}_k, \theta_{k-1})$
  end while
  return $\boldsymbol{x}_k$, θ_k

The proposed algorithms are described in Algorithms 2 and 3. `ODESolver` is a solver that does not compute the sensitivities of the state with respect to the parameters θ, thus, it is inherently faster. In the next section, we will demonstrate that Algorithm 3 is an approximation of Algorithm 1, where the difference comes from a redefinition of the residual function. Both algorithms follow naturally from a gradient descent algorithm applied to the optimization problem (1), reformulated to explicitly include equality constraints. The vector of optimization variables $\boldsymbol{x}_k$ generated by Algorithm 2 does not have to be an ODE solution except when it has converged to the local minimizer. By using the residual function $r(\boldsymbol{x}, \theta)$, in effect, we embed an ODE solver within the learning problem, without the need to explicitly solve ODEs. Such a formulation scales much better with the number of time samples, since we no longer have to solve, in a sequential manner, for the ODE solution and the state sensitivities.

Algorithm 3 Block coordinate gradient descent with residual functions based on implicit constraints and state reset.

Require: α_x, α_θ: Stepsizes
Require: $r(\boldsymbol{x}, \theta)$: Residual function as implicit dynamical constraints
Require: x_0: Initial state vector
Require: θ_0: Initial parameter vector
  $k \leftarrow 0$
  while θ_k, $\boldsymbol{x}_k$ not converge **do**
    $k \leftarrow k + 1$
    $\boldsymbol{x}_{k-1} \leftarrow \texttt{ODESolver}(\theta_{k-1}; x_0)$
    $\boldsymbol{x}_k \leftarrow \boldsymbol{x}_{k-1} - \alpha_x \nabla_{\boldsymbol{x}} F(\boldsymbol{x}_{k-1}, \theta_{k-1})$
    $\theta_k \leftarrow \theta_{k-1} - \alpha_\theta \nabla_\theta F(\boldsymbol{x}_k, \theta_{k-1})$
  end while
  return $\boldsymbol{x}_k$, θ_k

Accuracy of direct collocation method: Direct collocation methods minimize the error between the learned state trajectory and the actual ODE solution. Such local collocation methods are both computationally simple and efficient, and support batch executions. The collocation used in Algorithms 2 and 3 uses a third order polynomial to represent the ODE solution between two time instances. The resulting approximation error can be made smaller be choosing a finer discretization stepsize h. Alternatively, we can use higher order polynomials, e.g., fifth-order Gauss-Lobatto method described by Herman and Conway (1996). Such a method, based on Gauss-Lobatto quadrature (also known as Radau quadrature) is closely related to how ODE solvers compute solutions using the implicit Radau numerical solver presented in Hairer and Wanner (1999). In other words, rather than separately solving the ODE, we encode the ODE solver within the optimization problem via quadrature induced equality constraints, and jointly solve for both the parameter vector θ and the ODE solution. One advantage of the ODE solvers introduced in Chen et al. (2018) is the access to adaptive-step solvers. Residuals in Algorithm 3 can be formulated with an adaptive step, as well. Indeed, since we solve an ODE at each step, the solution can provide the step adaptive quadrature coefficients that can be used to construct the residual function. For example, the `scipy.integrate.solve_ivp` function can return an `OdeSolution` object that includes the time instants between which local interpolants are defined, and the local interpolants. These local interpolants can be used to construct residual functions and evaluate them at predetermined time instants.

Memory and time efficiency: The memory cost is a function of the number of optimization variables (states and parameters) and the number of the residuals imposing ODE solution constraints. Gradient evaluation depends only on the computational graph of $f(x; \theta)$. Unlike Algorithm 1, since there is no explicit dependence between the state variables (they are seen as independent optimization variables), state time causality has no explicit impact. The time complexity of Algorithms 2 and 3 depends on the rate of convergence of the gradient descent algorithm and the time per iteration. The rate of convergence for various versions of the gradient descent algorithm is discussed in Reddi

et al. (2018). The cost per iteration (epoch) plays a big factor in the total time to generate a solution. Algorithm 2 does not require solving ODEs, thus is the fastest.

Mini-batch executions: Algorithms 2 and 3 can be run on mini-batches based on time intervals: given a time window Δ, we can randomly select a time instant t_i and update the optimization variables based on a cost function defined using the optimization variables included in the mini-batch, only. In other words, the loss function is defined based on the state variables $\{x_i, \ldots, x_{i+\Delta}\}$. The only requirement is to have the cost function L separable in terms of time instants, requirement satisfied by a typical loss function such as the mean square error. In the case of Algorithm 3, one option is to solve the ODE for each epoch using the initial condition x_0, followed by gradient evaluations on mini-batches. Alternatively, we can reset the state by computing the solution of the ODE on the time interval corresponding to the mini-batch, where the initial state is chosen as x_i, the current estimate for the state at time instant t_i.

Loss function gradient: In the case of Algorithm 1, the gradient of the loss function depends on the sensitivity of the state with respect of the optimization variables $x_\theta = \frac{\partial x}{\partial \theta}$. In turn, the state sensitivity x_θ evolves according to the ODE $\dot{x}_\theta = \frac{\partial f}{\partial x} x_\theta + \frac{\partial f}{\partial \theta}$. Therefore, the stability of the gradient of the loss function depends on the Jacobian $\frac{\partial f}{\partial x}$. A poor choice of initial parameters θ can make the ODE unstable, leading to gradient explosion if the stepsize of the gradient descent algorithm is too small. Alternatively, a very stable ODE combined with a small $\frac{\partial f}{\partial \theta}$ can lead to fast gradient decay, hence a slow convergence of the gradient descent algorithm. Since both Algorithm 2 and 3 do not use explicitly the state sensitivity in the evaluation of the loss functions, their evolution will not be as significantly impacted by the dynamics of the state sensitivity.

Limitations: In Chen et al. (2018), the authors employ an ODE solver to generate solutions in the latent space for training a time series generative model. During each gradient update iteration, an initial condition in the latent space is randomly sampled from a Gaussian distribution, with its mean and covariance matrix being determined by a model that is continuously updated. This implies that the ODE solutions vary throughout the training process. However, we cannot apply our approach to this learning problem because, in addition to the various model parameters, the ODE solutions themselves are optimization variables. Therefore, they are assumed to remain consistent throughout the learning process.

Experiments: We evaluated Algorithms 1-3 on the problem of learning the parameters of the particle-based Cucker-Smale ODE model. Various versions of this model can be found in Carrillo et al. (2010). This model is nonlinear, and we can easily increase the state vector size by increasing the number of particles. We will compare the accuracy of the algorithms both on training and testing data, and the time per iteration (epoch). We also present loss function results based on the ODE solutions for parameter values explored during the search process. Such a loss function is more meaningful for comparison with Algorithm 1 since Algorithm 2 does not necessarily generates ODE solutions during the optimization process. We also demonstrate how our approach scales with the number of model parameters and its effectiveness in learning the parameters of a black-box model, one that lacks a specific structure, as seen in the case of the Cucker-Smale model. Furthermore, we illustrate how our proposed algorithms can be combined with algorithms employing sensitivity analysis enabled ODE solvers to speedup the learning process in an autoencoder training scenario.

2. Algorithms

Problem (1) can be reformulated as an optimization problem with equality constraints of the form

$$\begin{aligned} \min_{\theta,\boldsymbol{x}} \quad & \tilde{L}(\boldsymbol{x}) \\ \text{such that:} \quad & \tilde{r}(\boldsymbol{x},\theta)=0, \end{aligned} \tag{3}$$

where $\tilde{L}=L\circ\boldsymbol{y}$, $\tilde{r}(\boldsymbol{x},\theta)=\boldsymbol{x}-\hat{\boldsymbol{x}}(\theta)$ is the vector-valued residual function that quantifies how far $\boldsymbol{x}$ is from the solution of the ODE $\hat{\boldsymbol{x}}(\theta)=\texttt{ODESolver}(\theta;x_0)$. Assuming that $\tilde{L}(\boldsymbol{x})$ attains minimum value at zero, (3) is equivalent to the unconstrained optimization problem

$$\min_{\theta,\boldsymbol{x}} \tilde{L}(\boldsymbol{x})+\frac{1}{2}\|\tilde{r}(\boldsymbol{x},\theta)\|^2. \tag{4}$$

We later describe how we can deal with the case where $\tilde{L}(\boldsymbol{x})$ does not attain its minimum as zero. The unconstrained optimization problem (4) can be solved using a gradient descent algorithm, where the iterative equations are given by

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k-\alpha_{\boldsymbol{x}}\left[\nabla_{\boldsymbol{x}}\tilde{L}(\boldsymbol{x}_k)+\left(\frac{\partial\tilde{r}}{\partial\boldsymbol{x}}(\boldsymbol{x}_k,\theta_k)\right)^T\tilde{r}(\boldsymbol{x}_k,\theta_k)\right], \tag{5}$$

$$\theta_{k+1} = \theta_k-\alpha_{\theta}\left[\left(\frac{\partial\tilde{r}}{\partial\theta}(\boldsymbol{x}_k,\theta_k)\right)^T\tilde{r}(\boldsymbol{x}_k,\theta_k)\right], \tag{6}$$

where $\nabla_{\boldsymbol{x}}\tilde{L}(\boldsymbol{x}_k)=\left(\frac{\partial\boldsymbol{y}}{\partial\boldsymbol{x}}(\boldsymbol{x}_k)\right)^T\nabla_{\boldsymbol{y}}L(\boldsymbol{y}(\boldsymbol{x}_k))$. The Jacobians of the residual function $\tilde{r}$ can be explicitly written as:

$$\frac{\partial\tilde{r}}{\partial\boldsymbol{x}}=I,\ \frac{\partial\tilde{r}}{\partial\theta}=-\frac{\partial\hat{\boldsymbol{x}}}{\partial\theta},$$

where I is the identity matrix.

In the following, we recover Algorithm 1 by manipulating iterations (5)-(6). We make a first modification in (6) to bring it closer to the iteration (2). Namely, we replace $\boldsymbol{x}_k$ by $\boldsymbol{x}_{k+1}$ resulting in

$$\begin{aligned} \boldsymbol{x}_{k+1} &= \boldsymbol{x}_k-\alpha_{\boldsymbol{x}}\left[\nabla_{\boldsymbol{x}}\tilde{L}(\boldsymbol{x}_k)+\left(\frac{\partial\tilde{r}}{\partial\boldsymbol{x}}(\boldsymbol{x}_k,\theta_k)\right)^T\tilde{r}(\boldsymbol{x}_k,\theta_k)\right], \\ \theta_{k+1} &= \theta_k-\alpha_{\theta}\left[\left(\frac{\partial\tilde{r}}{\partial\theta}(\boldsymbol{x}_{k+1},\theta_k)\right)^T\tilde{r}(\boldsymbol{x}_{k+1},\theta_k)\right]. \end{aligned} \tag{7}$$

By replacing the residual function $\tilde{r}$ with the r defined in terms of implicit constraints, we have obtained Algorithm 2, which can be interpreted as iteration steps in a coordinate descent method, as described in Bertsekas (1999). This change is an intermediate step to get to Algorithm 3. After this change, $\boldsymbol{x}_k$ is not guaranteed to be a solution of the ODE. We can enforce this by resetting the state at each iteration to a solution of the ODE, i.e., $\boldsymbol{x}_k=\hat{\boldsymbol{x}}(\theta_k)$. This enforcement can be accomplished with any ODE solver, and does not have to compute the state sensitivities with respect to the vector of parameters. Thus, solving

the ODE is faster since no additional equations accounting for the state sensitivities are added. There are several effects of this second change. First, no term involving the residual function $\tilde{r}$ appears in (7), since $\tilde{r}(\boldsymbol{x}_k, \theta_k) = 0$. Second, the residual function evaluated at $(\boldsymbol{x}_{k+1}, \theta_k)$ becomes

$$\tilde{r}(\boldsymbol{x}_{k+1}, \theta_k) = -\alpha_{\boldsymbol{x}} \nabla_{\boldsymbol{x}} \tilde{L}(\boldsymbol{x}_k),$$

while the Jacobian $\frac{\partial \tilde{r}}{\partial \theta}(\boldsymbol{x}_{k+1}, \theta_k)$ remains unchanged since it does not depend on $\boldsymbol{x}$, i.e.,

$$\frac{\partial \tilde{r}}{\partial \theta}(\boldsymbol{x}_{k+1}, \theta_k) = \frac{\partial \hat{\boldsymbol{x}}}{\partial \theta}(\theta_k).$$

Therefore, we obtain the new iterations

$$\begin{aligned} \boldsymbol{x}_k &= \texttt{ODESolver}(\theta_k; x_0), \\ \theta_{k+1} &= \theta_k - \alpha \left[\left(\frac{\partial \hat{\boldsymbol{x}}}{\partial \theta}(\theta_k) \right)^T \nabla_{\boldsymbol{x}} \tilde{L}(\boldsymbol{x}_k) \right]. \end{aligned} \tag{8}$$

where $\alpha = \alpha_\theta \alpha_{\boldsymbol{x}}$ and (8) is exactly the iteration (2). Hence, unsurprisingly, we have recovered the gradient descent algorithm for solving (1). It should be clear by now that to avoid having to explicitly compute the sensitivities of the state vector with respect to θ, we change the residual function $\tilde{r}$ by another residual function that does not have such requirements. An example of such a residual function is based on the Hermite-Simpson derivative collocation with trapezoid quadrature, discussed in Hargraves and Paris (1987), and is given by

$$r(\boldsymbol{x}, \theta) = -\boldsymbol{x}_{1:N} + \boldsymbol{x}_{0:N-1} + \frac{h}{6} \left[f(\boldsymbol{x}_{0:N-1}; \theta) + f(\boldsymbol{x}_{1:N}; \theta) + 4 f(\boldsymbol{x}_c; \theta) \right],$$

where $\boldsymbol{x}_c = \frac{1}{2} \left[\boldsymbol{x}_{1:N} + \boldsymbol{x}_{0:N-1} \right] + \frac{h}{8} \left[f(\boldsymbol{x}_{0:N-1}; \theta) - f(\boldsymbol{x}_{1:N}; \theta) \right]$. The residual function is zero when evaluated at a solution of the ODE. The evaluation of this residual function is very efficient since batches of time instances can be executed in parallel.

Assuming Lipschitz continuity of $f(x)$, we are guaranteed by the Picard–Lindelöf theorem found in Lindelöf (1894) that the ODE has a solution and it is unique. Hence by replacing the residual function $\tilde{r}(\boldsymbol{x}, \theta)$ with $r(\boldsymbol{x}, \theta)$, we do not change the solution of (3). Retracing the previous steps, where instead of using the residual function $\tilde{r}$, we use r, we recover Algorithm 3. We note that since $\boldsymbol{x}_k$ is a solution of the ODE, $r(\boldsymbol{x}_k, \theta_k) = 0$, and therefore $\nabla_{\boldsymbol{x}} F(\boldsymbol{x}_k, \theta_k) = \nabla_{\boldsymbol{x}} \tilde{L}(\boldsymbol{x}_k)$. The gradients and Jacobians that appear in Algorithm 2 and Algorithm 3 can be computed using automatic differentiation.

2.1 Enforcing the Residual Function Equality Constraints

If $\tilde{L}$ does not attain its minimum at zero, we need to make sure that the norm of the residual function is forced to be as small as possible. We can achieve this by introducing a hyper-parameter λ that acts as a weight for the residual function. The new cost function becomes $F(\boldsymbol{x}, \theta) = \tilde{L}(\boldsymbol{x}) + \frac{\lambda}{2} \|r(\boldsymbol{x}, \theta)\|^2$ and we can do a hyper-parameter search to reduce the magnitude of the residual function. A better approach is to update λ during the optimization process, based on the magnitude of the residual function. A guided strategy for updating online the weight function is based on a variant of the Augmented

Lagrangian method introduced in Hestenes (1969) that employs the extended Lagragian function: $\mathcal{L}_\rho(\boldsymbol{x}, \theta, \lambda) = L(\boldsymbol{x}) + \lambda^T r(\boldsymbol{x}, \theta) + \frac{\rho}{2}\|r(\boldsymbol{x}, \theta)\|^2$, for $\rho > 0$. In this method, we solve a min-max optimization problem: $\min_{\boldsymbol{x},\theta} \max_\lambda \mathcal{L}(\boldsymbol{x}, \theta, \lambda)$. An extension of Algorithm 2 that enforces the residual equality constraint is summarized in Algorithm 4.

Algorithm 4 Augmented Lagrangian method based on gradient descent iterations

Require: α_x, α_θ: Stepsizes
Require: $r(\boldsymbol{x}, \theta)$: Residual function as implicit dynamical constraints
Require: x_0: Initial state vector
Require: $\rho > 0$: Residual function norm weight
Require: θ_0: Initial parameter vector
  λ_0: Initial Lagrange multiplier vector
  $\boldsymbol{x}_0 \leftarrow \texttt{ODESolver}(\theta_0; x_0)$
  $k \leftarrow 0$
  while θ_k, $\boldsymbol{x}_k$, λ_k not converge **do**
    $k \leftarrow k + 1$
    $\boldsymbol{x}_k \leftarrow \boldsymbol{x}_{k-1} - \alpha_x \nabla_{\boldsymbol{x}} \mathcal{L}_\rho(\boldsymbol{x}_{k-1}, \theta_{k-1}, \lambda_{k-1})$
    $\theta_k \leftarrow \theta_{k-1} - \alpha_\theta \nabla_\theta \mathcal{L}_\rho(\boldsymbol{x}_k, \theta_{k-1}, \lambda_{k-1})$
    $\lambda_k \leftarrow \lambda_{k-1} + \rho r(\boldsymbol{x}_k, \theta_k)$
  end while
  return $\boldsymbol{x}_k$, θ_k

3. Experiments

We evaluated Algorithms 1 - 3 on the Cucker-Smale model describing the interaction between particles that include self-propelling, friction and attraction-repulsion phenomena. It takes into account an alignment mechanism of the particles by averaging their relative velocities with all the other particles. The strength of this averaging process depends on the mutual distance. For N particles the model is described by the following dynamical system:

$$\begin{aligned} \dot{x}_i &= v_i, \\ \dot{v}_i &= \frac{1}{N}\sum_{j=1}^{N} H(\|x_i - x_j\|)(v_j - v_i) - \frac{1}{N}\nabla U(\|x_i - x_j\|), \end{aligned}$$

for $i = 1, \ldots, N$, where x_i and v_i are the two-dimensional particle positions and velocities, $H(r) = \frac{1}{(1+r^2)^\gamma}$ is the communication rate, and $U(r) = -c_a e^{-r/l_a} + c_r e^{-r/l_r}$ is the inter particle potential energy. The parameter γ is a positive scalar, and c_a, c_r and l_a, l_r are the strength and the length of the attraction and repulsion, respectively. The number of states is linear in the number of particles, i.e., $4N$. The learning problem is defined as follows: given the particle trajectories (i.e., time series of positions and velocities) learn the parameters of model.

We tested the algorithms for various number of particles: $N \in \{5, 10, 20, 50, 100, 200\}$. We used one time series as training data, generated with the same, random, initial parameters, for all cases. The training loss function is the sum of squared errors (SSE). The test

data consists of model trajectories generated with random initial conditions. We compared Algorithms 1 - 3 using four metrics: average iteration time, training loss function (i.e., SSE), training relative sum of squared error (RSSE) based on ODE solutions, and RSSE on test data. We run the algorithm for 5k epochs and evaluated the four metrics for all combinations of algorithms and number of particles. The RSSE metric is computed with respect to the solution of the ODE for a particular instance of the model parameters generated during the optimization process. This way we can compare how far the solution of the ODE is from the target trajectories. The total optimization time is computed as the product between the average epoch time and the number of epochs. Since Algorithms 1 and 3 must solve ODEs, their epochs are computationally more expensive. We implemented the three algorithms on both Pytorch and Jax and used their ODE solvers to generate ODE solutions and state sensitivities, when needed. In particular, we used the adaptive step `Dopri5` ODE solver. The Jax implementation is faster since we took advantage of the just in time compiling (JIT) feature. The training and testing were done on a PC with Intel 12 core Xeon 3.5 GHz CPU with 64 GB of RAM, and an NVIDIA GEFORCE RTX 2080 Ti GPU card.

3.1 Pytorch Implementation

The Pytorch implementation details of the three algorithms are shown in Table 1. In

	State update	**Parameter update**	ODE solver
Algorithm 1	X	Adam ($l_r = 0.01$)	Dopri5
Algorithm 2	SGD ($l_r = 0.01$)	Adam ($l_r = 0.01$)	X
Algorithm 3	SGD ($l_r = 1$)	Adam ($l_r = 0.01$)	Dopri5

Table 1: Algorithms 1—3 Pytorch implementation details.

the case of Algorithm 2 we used SGD and Adam algorithms for updating the state and the parameters, respectively. Both algorithm used a stepsize $l_r = 0.01$. In the case of Algorithm 3, we used the same combination of algorithms, but the stepsize for SGD is $l_r = 1$, so that the product between the two stepsizes in 0.01. All experiments were executed on a GPU device. We used the ODE solver in the `torchdiffeq` library for Algorithms 1 and 3. In the case of Algorithm 1 we tested the performance of both direct and adjoint methods for the sensitivity analysis. In what follows, we refer to optimization algorithms that use the direct and adjoint methods, as *Algorithm 1, direct*, and *Algorithm 1, adjoint*, respectively. Both options are provided by the `torchdiffeq` library. In the case of Algorithm 3, the ODE solver was used to compute the ODE solution only, and not the state sensitivities. Table 2 shows the epoch time for each of the three algorithms for various number of particles. As expected, Algorithm 3 average iteration time is smaller than Algorithm 1 iteration time, since no sensitivities are computed. Algorithm 2 is at least 4x faster then Algorithm 1 since it makes the best use of the parallel computations enabled by the GPU. The GPU was used for all experiments related to Algorithm 1 resulting in rather flat average iteration time. Slight improvements in time per epoch can be obtain if the CPU is used for small number of particles. However, CPU usage does not scale with the number of particles. Not unexpectedly, the implementation that uses the adjoint method is much slower since the number of state variables dominate the number of parameters.

	N=5	N=10	N=20	N=50	N=100	N=200
Algorithm 1, direct	0.463	0.602	0.594	0.565	0.606	0.614
Algorithm 1, adjoint	2.655	2.651	2.665	2.673	2.681	2.819
Algorithm 2	0.009 (50x)	0.008 (75x)	0.008 (74x)	0.011 (51x)	0.037 (16x)	0.155 (4x)
Algorithm 3	0.171 (3x)	0.188 (3x)	0.192 (3x)	0.179 (3x)	0.184 (3x)	0.279 (2x)

Table 2: Pytorch: Algorithms 1—3 average time per epoch in seconds. Improvements of Algorithms 2 and 3 over the fastest version of Algorithm 1 are shown in parentheses.

Figure 1 shows the training loss for the three algorithms in logarithmic scale. Only the training losses for Algorithms 2 and 3 are truly comparable since they have the same loss function. We note that Algorithm 1's loss function at the end of training is comparable to that of Algorithm 3. The results show that Algorithm 2 has the best performance in training, measured by the SSE value and the time it needed to reach the SSE value, and it is followed by Algorithm 3. Algorithm 1 has the worst performance in training, however since the loss function is defined with respect to a different residual function, we cannot draw a definitive conclusion. More insights on the performance of the three algorithms come from comparing the RSSE plots. The RSSE metrics are computed based on the ODE solutions at the model parameter instances generated during the optimization process. Figure 2 shows the RSSE plots of the three algorithms for various numbers of particles. All observations we made about the SSE plots carry to this set of plots. Algorithms 2 and 3 are superior to Algorithm 1 (both versions) in convergence time and best achieved RSSE value. We evaluated the accuracy of the results against test data generated using 100 randomly selected initial conditions. The average RSSE metrics for each case, are shown in Table 3. Similar to the training results, Algorithm 2 gives the most accurate results. Next most accurate results are generated by Algorithm 3, followed by Algorithm 1.

	N=5	N=10	N=20	N=50	N=100	N=200
Algorithm 1, direct	3.97×10^{-5}	2.42×10^{-6}	1.82×10^{-6}	7.26×10^{-7}	5.00×10^{-7}	4.32×10^{-7}
Algorithm 1, adjoint	3.98×10^{-5}	2.44×10^{-6}	1.85×10^{-6}	7.40×10^{-7}	5.12×10^{-7}	4.41×10^{-7}
Algorithm 2	1.65×10^{-6}	2.50×10^{-7}	1.77×10^{-7}	8.52×10^{-8}	3.70×10^{-8}	3.24×10^{-8}
Algorithm 3	3.65×10^{-6}	1.85×10^{-7}	3.18×10^{-7}	1.96×10^{-6}	2.37×10^{-7}	1.98×10^{-7}

Table 3: Pytorch: Algorithms 1—3 RSSE on test data.

3.2 Jax Implementation

We repeated the experiments using Jax implementations. One advantage using Jax is the speedups enabled by JIT. The optimization algorithms implementation details are shown in Table 4. In the case of Algorithm 2, for stability reasons, we changed the stepsizes for the SGD and Adam algorithm, while making sure that their product is equal to the stepize product used in the Pytorch implementation, i.e., 0.01. The times per iteration in the Jax implementation are shown in Table 5. Similar to the Pythorch implementation, Algorithms 2 and 3 are much faster than Algorithm 1. In addition, the training losses for Algorithms 2 and 3, shown in Figures 3 and 4, are consistently smaller than Algorithm 1's training loss. Algorithm 1 uses the Jax provided ODE solver with the same `Dopri5` integration scheme, as in the Pytorch case. The sensitivity analysis in the ODE solver implementation is based on the adjoint method. We note that even when using JIT, the time per iteration

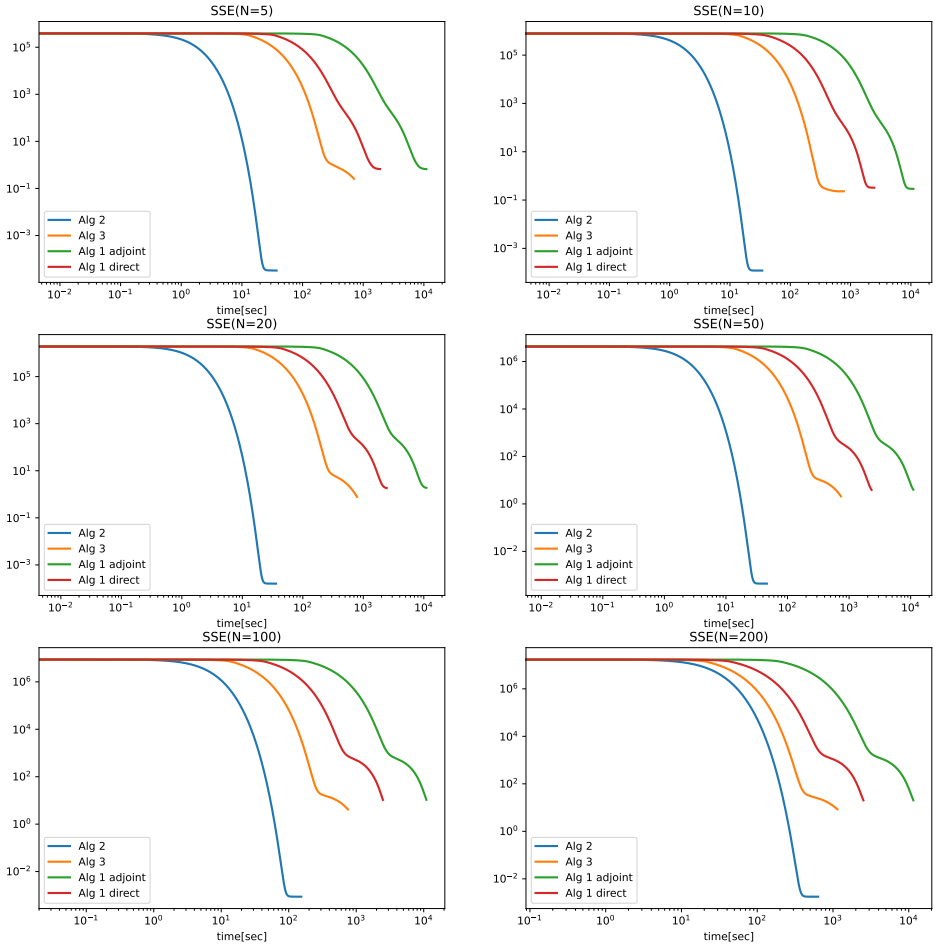


Figure 1: Pytorch: SSE training loss comparison for various number of particles.

	State update	**Parameter update**	ODE solver
Algorithm 1	X	Adam ($l_r = 0.01$)	Dopri5
Algorithm 2	SGD ($l_r = 0.01$)	Adam ($l_r = 0.01$)	X
Algorithm 3	SGD ($l_r = 0.1$)	Adam ($l_r = 0.1$)	Dopri5

Table 4: Algorithms 1—3 Jax implementation details.

of the Jax implementation of Algorithm 1 is significantly slower than that of the Pytorch implementation using the direct method, but much faster that the adjoint version. In the case of Algorithms 2 and 3, the time efficiency is dramatically better in Jax than in Pytorch, mainly due to the use of the JIT feature.

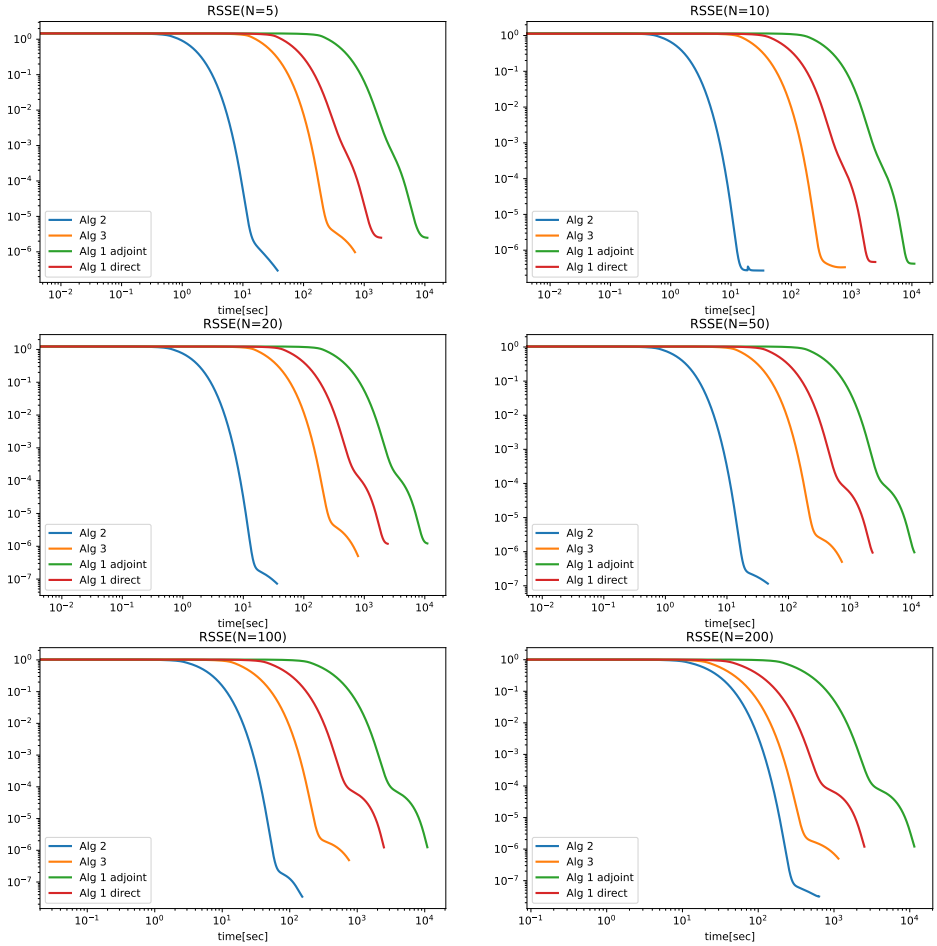


Figure 2: Pytorch: RSSE training loss comparison for various number of particles.

	N=5	**N=10**	**N=20**	**N=50**	**N=100**	**N=200**
Algorithm 1, adjoint	0.935	0.973	1.243	1.526	1.728	1.896
Algorithm 2	0.0012 (x780)	0.0014 (x695)	0.0015 (x827)	0.0013 (x1173)	0.0014 (x1234)	0.0013 (x1458)
Algorithm 3	0.06 (x16)	0.06 (x16)	0.061 (x20)	0.062 (x25)	0.065 (x27)	0.072 (x27)

Table 5: Jax: Algorithms 1—3 average time per epoch in seconds. Improvements of Algorithms 2 and 3 over Algorithm 1 are shown in parentheses.

We run the models learned using Algorithms 1, 2 and 3 on the same test data used in the Pytorch case. The results are shown in Table 6, showing that Algorithm 2 fares better than both Algorithms 1 and 3, while Algorithm 3 is comparable in accuracy to Algorithm 1. Note that Algorithm 3 exhibits an oscillatory behavior near the local minima. Such a behavior

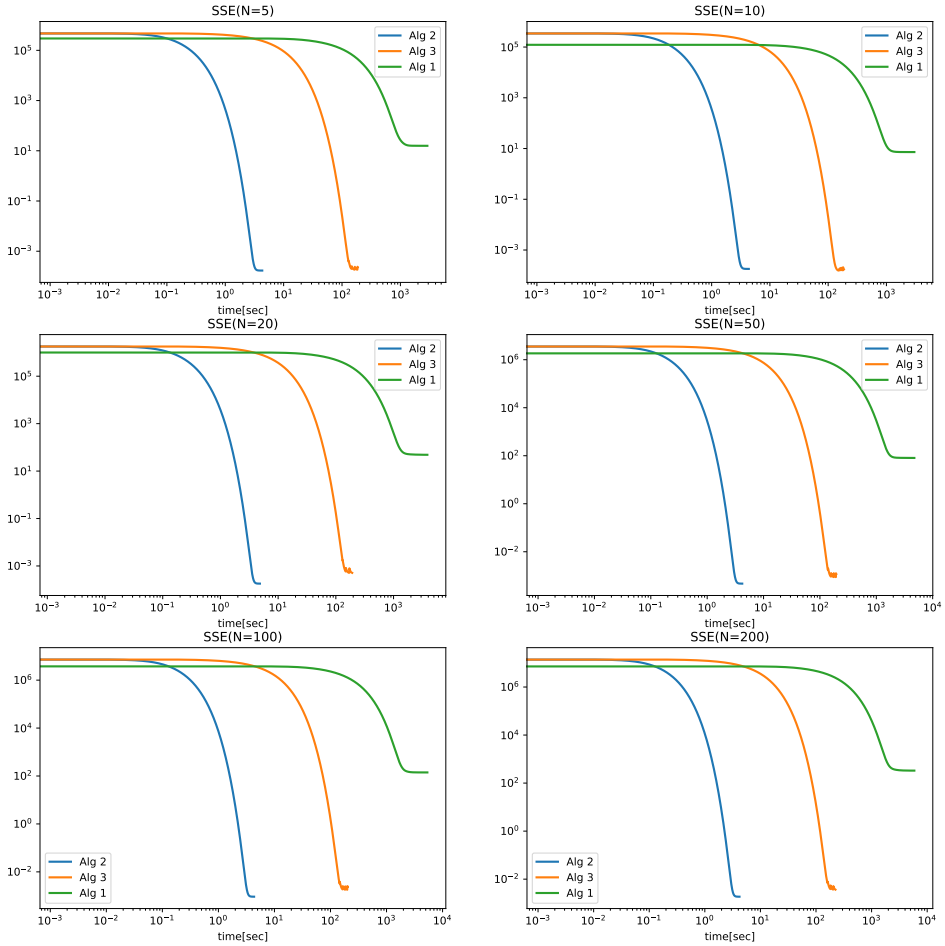


Figure 3: Jax: SSE training loss comparison for various number of particles.

is less visible in the Pytorch case. We speculate that this behavior is induce, in part, by the ODE solver implementations in Jax, combined with the adaptive nature of the gradient scaling in Adam algorithm. An additional source of the oscillating behavior near the local minima is the approximation error induced by the use of collocation methods to approximate the ODE dynamics; error that can be reduced by using a finer time discretization scheme.

3.3 Sensitivity to the Initial Conditions of the Optimization Variables

We present experimental results demonstrating the sensitivity of the optimization algorithms to the initial values of the optimization variables. These results and the results

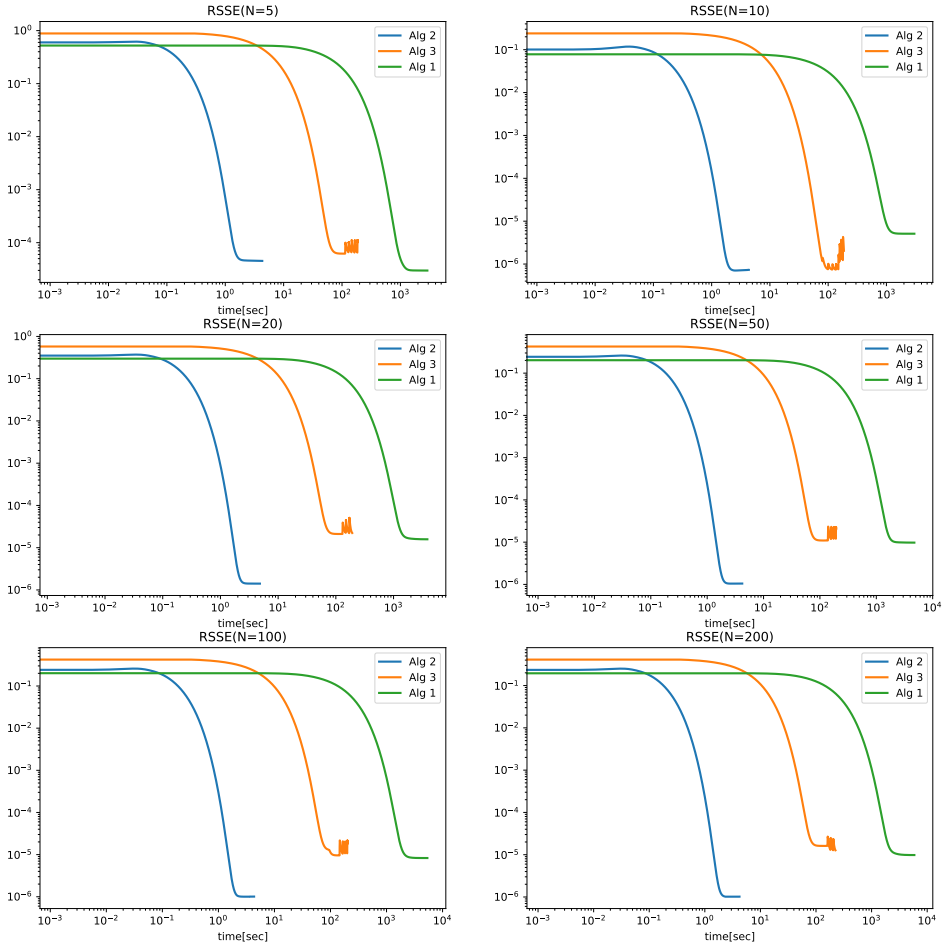
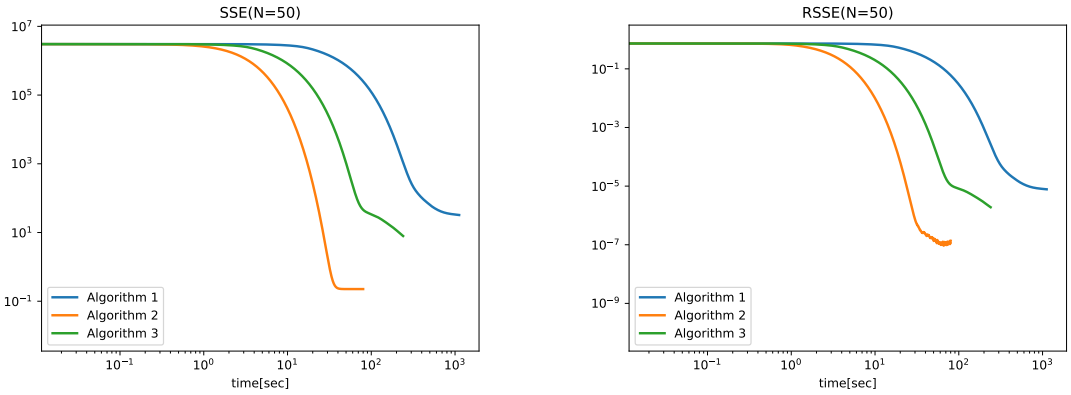


Figure 4: Jax: RSSE training loss comparison for various number of particles.

	N=5	**N=10**	**N=20**	**N=50**	**N=100**	**N=200**
Algorithm 1	9.78×10^{-5}	5.31×10^{-5}	2.43×10^{-5}	2.10×10^{-5}	1.94×10^{-5}	1.94×10^{-5}
Algorithm 2	7.78×10^{-5}	6.74×10^{-7}	1.86×10^{-7}	1.40×10^{-7}	1.56×10^{-7}	1.46×10^{-7}
Algorithm 3	9.52×10^{-5}	2.29×10^{-6}	3.02×10^{-5}	2.43×10^{-5}	2.25×10^{-5}	2.17×10^{-5}

Table 6: Jax: Algorithms 1—3 RSSE on test data.

in the following sections are generated using Pytorch implementations. Specifically, we examine this phenomenon in the context of 50 particles, comparing the performance of Algorithms 1, 2 and 3. We use only the *direct* version of Algorithm 1. We sample random initial values for the optimization variables from a uniform distribution, and we conduct



(a) SSE training loss.

(b) RSSE training loss.

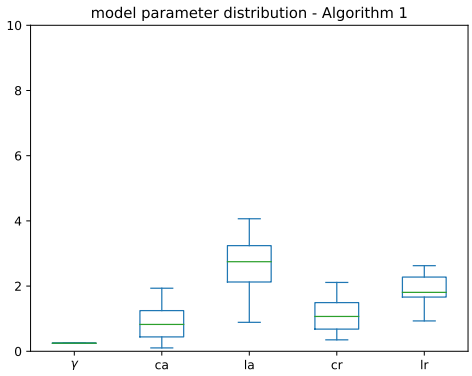
Figure 5: Uncertainty quantification of loss metrics under random initial conditions for the optimization variables. Dark color curves represent the mean metrics, while light color areas is determined by the variances of the loss functions, over the random trials.

10 trials to learn the optimal parameters of the Cucker-Smale model. Our report includes information on the uncertainty surrounding the SSE and RSSE metrics, as well as the distributions of the learned parameters. We employ similar hyperparameters for the optimization algorithms as outlined in the previous section. Figures 5 illustrate the impact of random initial conditions on the uncertainty in the SSE and RSSE metrics. Notably, Algorithm 3 appears to exhibit higher sensitivity, in both metrics. Algorithm 2 has a rather small uncertainty on the SSE loss. Although our metrics exhibit more variability in the training results, the variability is concentrated at the lower values of the loss functions. Moreover, the loss metrics of Algorithms 2 and 3 are not worse than the loss of Algorithm 1, even when considering this variability. Note that both axes of the plots are logarithmic, thus when plotted on decimal scale, the uncertainty of the loss functions would appear insignificant.

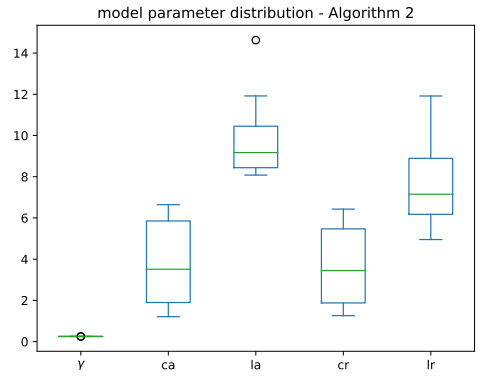
Figure 6 illustrates the uncertainty in the optimization variables after the final training iteration, using box plots. With the exception of parameter γ, that is correctly estimated by all algorithms, the remaining parameters do exhibit variability in their learned values. However, this is not necessarily a surprise, since many values for the coefficients in the exponential terms of the Cucker-Smale model can generate similar trajectories. This phenomenon is clear in parameter l_a. Once l_a is large enough, increasing its value will produce no significant changes in predictions . Figure 6d shows the histograms of the RSSE metric on 100 test trajectories with random initial conditions, and over the trained models corresponding to the 10 trials. Algorithms 2 and 3 tend to outperform Algorithm 1, in the majority of the tests.

3.4 Increasing the Number of Parameters in the Cucker-Smale Model

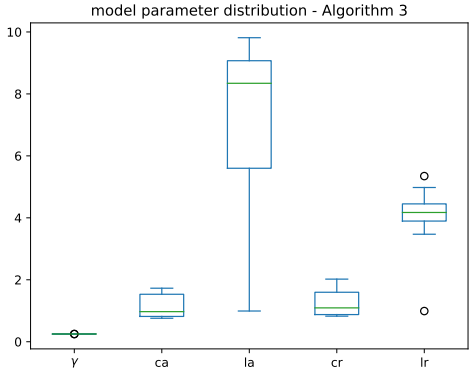
In the preceding section, we presented various statistics pertaining to the training algorithms' performance in learning five parameters of the Cucker-Smale model. In this section, we present results obtained when the number of parameters scales quadratically with the



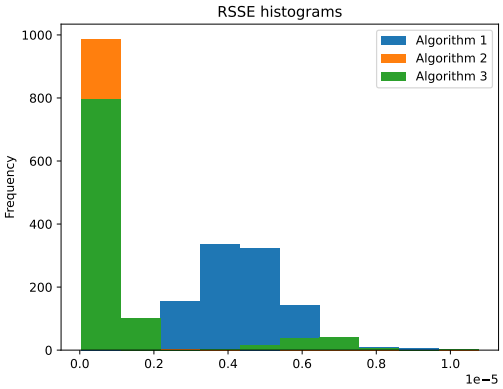
(a) Algorithm 1.



(b) Algorithm 2.

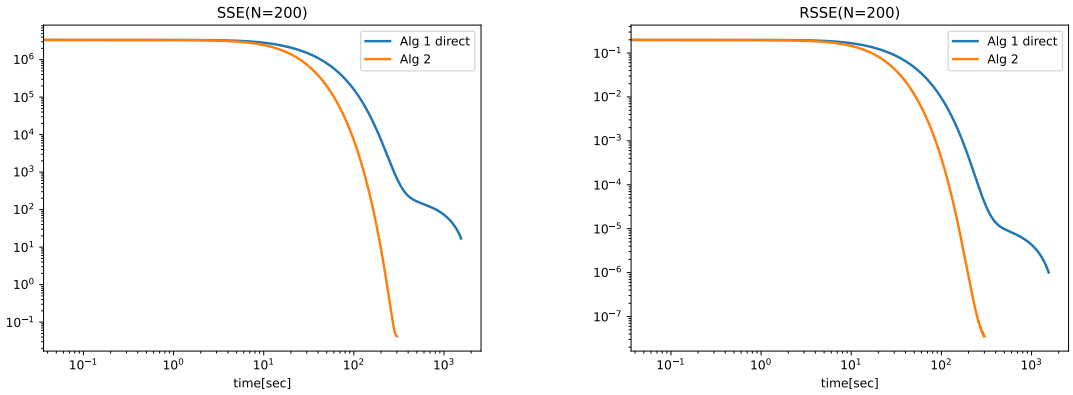


(c) Algorithm 3.



(d) RSSE histogram for Algorithms 1, 2 and 3, over 10 trials and 100 test trajectories with random initial conditions.

Figure 6: Uncertainty quantification: (i) optimization variables boxplots after the last training iteration and (ii) RSSE histograms on test data.



(a) SSE training loss comparison.

(b) RSSE training loss comparison.

Figure 7: Training results for 200 particles and Cucker-Smale model with non-uniform γ parameters.

number of agents of the model. We achieve this by introducing a modification to the γ parameter in the definition of the communication rate, making it dependent on the interaction between pairs of agents. Consequently, the γ parameter becomes $\gamma_{i,j}$, where i and j represent two distinct agents. For a model with $N = 200$ agents, this leads to a total of $N(N-1)/2 = 4950$ γ parameters.

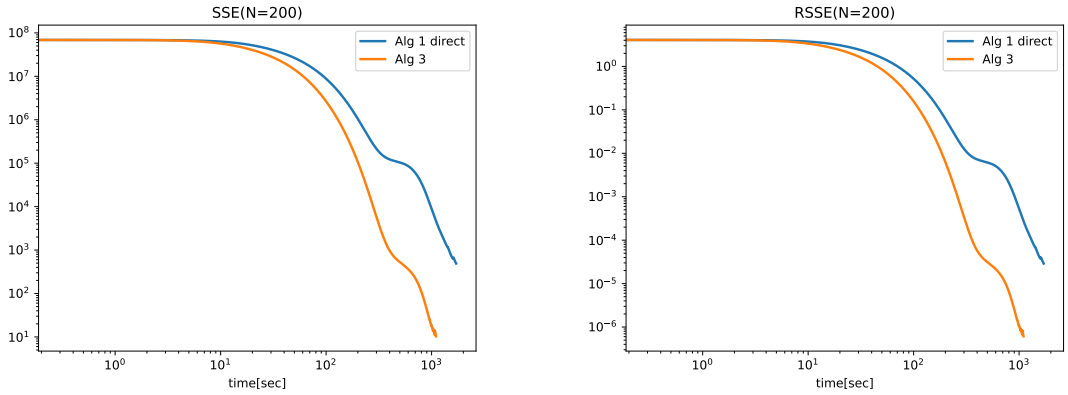
We examine the performance of two algorithms: Algorithm 2, which employs block coordinate gradient descent without state reset, and Algorithm 1, which utilizes gradient descent with a sensitivity-enabled ODE solver. We investigate the scenario where $N = 200$, and the details of the training experiments are provided in Table 7. Note that Algorithm 2 exhibits a computational speed advantage, being nearly 100 times faster than Algorithm 1.

	Algorithm 1 direct	**Algorithm 2**
# iterations	5000	5000
optim alg	Adam	Adam
learning rate	0.005	0.005
ODE solver	Dopri5	X
avg iter time	0.45 sec	0.06 sec

Table 7: Cuker-Smale model training experiment for 200 particles and with non-uniform γ parameters.

The decay of the SSE and RSSE metrics during training is depicted in Figures 7. As expected, Algorithm 2 is faster and generates lower training losses by the end of the training process.

We conducted tests on both models by comparing them against ground truth, test trajectories generated around the initial condition of the trajectory used for training. Specifically, we employed a Gaussian distribution centered at the (random) initial condition used to generate the training data, with a variance of 5. The objective is to assess the performance of the two trained models under similar conditions. Algorithm 2 yielded an average RSSE of



(a) SSE training loss comparison.

(b) RSSE training loss comparison.

Figure 8: Training results for 200 particles and black box models.

approximately 3.6×10^{-4} over 100 trials, while Algorithm 1 produced an average RSSE of approximately 3.5×10^{-4}. Although Algorithm 1 exhibited slightly better generalization with a 10^{-5} improvement, it operated at nearly 100 times slower speed. This disparity in time performance becomes even more pronounced when implementing Algorithm 2 using the JIT feature, as demonstrated in previous experiments.

3.5 Learning Black-Box models for the Cucker-Smale Dynamics

In this section, we conduct a comparative analysis of Algorithms 1 and 3 in the context of learning black box models for the Cucker-Smale dynamics. We are specifically focusing on a scenario where $N = 200$, resulting in a state dimension of 800. To represent the right-hand side of the ODE, we employ a neural network with two hidden layers, each of which is designed to match the state vector dimension, i.e., 800. We utilize the `tanh` activation function for these layers. The training data used is consistent with that of the previous sections. Table 8 provides an overview of the training experiment details, while Figures 8 illustrate the training results of the black-box models. The training process was executed on a GPU. When examining both the SSE and RSSE metrics, it becomes evident that Algorithm 3 outperforms Algorithm 1 in terms of absolute training time and loss metrics.

	Algorithm 1 direct	**Algorithm 3**
# iterations	4000	4000
optim alg	Adam	Adam
learning rate	0.0001	0.0001
ODE solver	Dopri5	Dopri5 (state reset only)
avg iter time	0.45 sec	0.29 sec

Table 8: Black box model training experiment for 200 particles.

We conducted tests on the two models by comparing them against ground truth trajectories generated by sampling around the initial conditions of the training trajectory. Specifically, we employed a Gaussian distribution centered at the initial condition used to

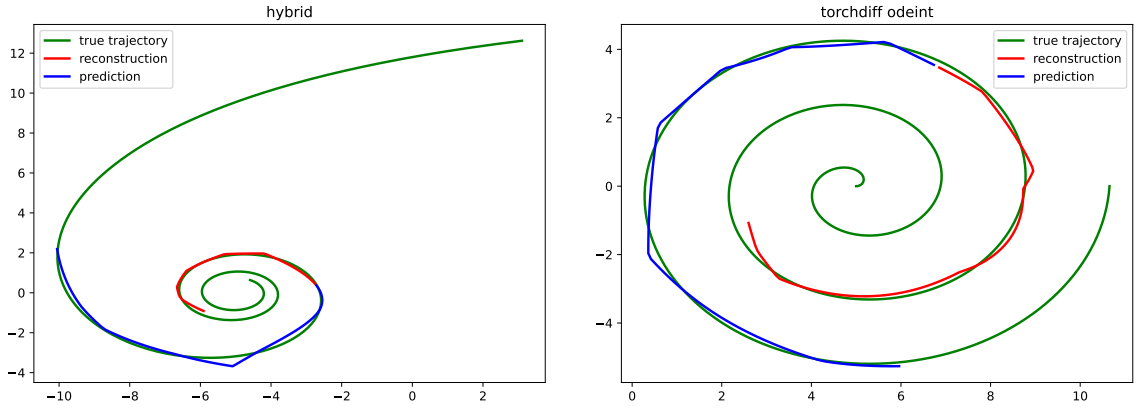
generate the training data, with a variance of 1. Given the high number of parameters in the black-box model, we do not anticipate that the models will generalize well far from the training trajectory. To enhance model generalization, we could consider utilizing a more extensive dataset comprising numerous trajectories originating from various initial coordinates as part of the training data. However, our primary objective in this context is to evaluate the performance of the two trained models under similar conditions. Both algorithms yielded an average RSSE metric of 0.111 when computed across 100 test trials. This experiment demonstrates that, while there is no noticeable improvement in the test data performance metric, Algorithm 3 exhibits greater speed and the potential for even greater speed when utilizing the JIT feature.

3.6 A Hybrid Approach for Speeding up Learning Dynamical Models

The time needed by learning algorithms to converge to a (local) minimizer depends on various factors, including the type of optimization algorithm (whether it is gradient-based, gradient-free, or of the first or second order), problem complexity, and distance between the initial values of the optimization variables and a local minima. While our proposed algorithms are faster since they do not explicitly perform sensitivity analysis, they do introduce approximation errors because they approximate ODE solutions using collocation methods. We propose a hybrid approach in which we employ Algorithm 3 to compute an initial estimate of the optimization variables. This estimate is then refined using Algorithm 1, which includes a sensitivity-enabled ODE solver. We compare the hybrid approach with the scenario where only Algorithm 1 is used. We use training time and prediction accuracy on test data as metrics for comparison. As an example, we adapted the approach for training time series generative models shown in Chen et al. (2018). The authors investigated the ability of ODE models to extrapolate time series. The proposed architecture includes a recurrent neural network (RNN) with 25 hidden units in the recognition network to output the initial value of the latent trajectory, which is 4-dimensional. The latent space dynamics are modeled with a one-hidden-layer network with 20 hidden units, and the decoder is a neural network with one hidden layer containing 20 hidden units. Unlike the original implementation, we learn an autoencoder instead of a variational autoencoder (VAE) to avoid dealing with sample-based ODE solutions. We kept the architectures of the various neural networks as described in Chen et al. (2018), except for the dimension of the output of the RNN, which is 4 in our case, since we do not output a mean and variance. Additionally, instead of optimizing the ELBO cost function typically used in VAE problems, we minimize the MSE cost. We set the maximum number of iterations to 4000, and use Adam, with a constant learning rate of 0.001, as optimization algorithm. We generated a dataset of 6000, 2-dimensional spirals, each starting at a different point, sampled at 200 equally-spaced time steps. We use the first 3000 for training, and the remaining 3000 for testing.

For the hybrid scenario, we split the training process in two steps. We use Algorithm 3 for the first 1250 iterations, after which we switch to Algorithm 1, for the remaining 2750 iterations. Algorithm 1 runs a sensitivity-enabled ODE solver implementing the direct method.

To test the trained models we use the first half of the time samples of the test trajectories for reconstructing the initial conditions of the latent trajectories. We use these initial



(a) Hybrid based approach (b) ODE solver based approach

Figure 9: Examples of reconstructed and predicted trajectories generated by models trained using the hybrid approach and the approach that uses a sensitivity analysis enabled ODE solver.

conditions to simulate the latent dynamics and produce trajectories in the latent space. These latent trajectories are then passed through the decoder to reconstruct and predict trajectories. The results are show in Table 9. Not unexpectedly, the hybrid approach is

	Hybrid	**`torchdiffeq` odeint**
avg iter time	0.31 sec	0.43 sec
total training time	1262 sec	1722 sec
MSE training	0.0164	0.05
RMSE reconstruction	0.175	0.320
RMSE prediction	0.241	0.213

Table 9: Comparison between the hybrid and the sensitivity-enabled ODE solver-based (i.e., torchdiffeq) AE training and testing results.

faster since one iteration of Algorithm 3 is roughly 3x faster than one iteration of Algorithm 1. The hybrid approach provides a smaller training cost at the end of the training process, and a smaller average reconstruction cost on test data. The implementation based on Algorithm 1 does offer a better average prediction cost on test data. Examples of reconstructions (red color) and predictions (blue color) of trajectories are shown in Figure 9. Note that we do not reconstruct/predict the full trajectories, but a total of 200 samples: 100 samples for reconstruction, and the next 100 samples for prediction. This example shows the potential for significant training time reductions by first using a faster, but approximate optimization algorithm to get close to a local minima. In other words, Algorithm 3 acts as a faster, surrogate of Algorithm 1.

4. Related Work

The authors of Chen et al. (2018) introduced the notion of "neural" ODE, where they parameterized the right-hand side f of an ODE by a neural network. Their approach for computing gradients of loss functions uses the adjoint method for implementing sensitivity analysis, and avoids using backpropagation. Thus, it eliminates the memory cost and numerical errors induced by differentiating through the operations of the forward pass. Chen et al. (2018) made available a Pytorch library named `torchdiffeq` that enables compositions of neural-ODEs with Pytorch layers. Moreover, the parameters of the resulting model can be trained simultaneously. The library `torchdiffeq` includes both direct and adjoint methods to support sensitivity analysis. Chen et al. (2018); Rubanova et al. (2019); Grathwohl et al. (2019) use neural ODEs to learn latent time series models, density models, and as a replacement for very deep neural networks. These algorithms are closely related to the sensitivity analysis methods introduced in Gardner et al. (2022); Hindmarsh et al. (2005). They include explicit ODE solvers only since they do not require Newton-Raphson steps to solve the implicit nonlinear equations. Our approach to integrating dynamical constraints described in Algorithms 2 and 3 is compatible with Pytorch models as well. Parameter training would require adding a loss function for minimizing the ODE-induced residuals. When using the state-reset approach, we can use any ODE solver, implicit or explicit. Jax includes the ability to solve ODEs as well, and it does include the `Dopri5` solver we used for comparison purposes. In the current implementation, the state sensitivities are computed using adjoint methods only. The Jax based ODE solver library `Diffrax` described in Kidger (2021) supports sensitivity analysis computations, as well. This library includes a similar number of integrators as `torchdiffeq`. One of the advantages of Jax is the ability to use the JIT decorator that results in tremendous computational efficiency gains. There are efforts to provide similar capability to Pytorch, however these efforts have not yet resulted in the same level of maturity as in the Jax case. Kidger et al. (2021) demonstrate a reduction in function evaluations, potentially reaching up to a 60% decrease, in the execution of the adjoint method for sensitivity analysis. This improvement is achieved by substituting the conventional L^2 norm in the backpropagation gradient update with a semi-norm. In a complementary vein, Zhuang et al. (2020) present an alternative approach to enhance the numerical efficiency of backpropagation gradient updates executed as part of the adjoint method for sensitivity analysis. The authors delve into a potential explanation for the suboptimal performance of neural-ODE based learning techniques compared to discrete layer methods. Their hypothesis is rooted in the presence of numerical errors within the reverse-mode integration process of the adjoint method. To address this issue, they introduce the adaptive checkpoint adjoint (ACA) method. This technique incorporates a trajectory checkpoint strategy to ensure the accuracy of the reverse-mode trajectory. It eliminates the redundant components in the shallow computation graphs generated by backpropagation. While discrete layer models may offer superior performance, they necessitate uniform time discretization. Neural-ODEs, on the other hand, can be particularly advantageous for handling non-uniformly sampled data, which is often encountered in sensor measurement time series. The proposed Algorithms 2 and 3 avoid having to expletively evaluate differential equations to implement the sensitivity analysis, employing whether direct or adjoint methods. However, these algorithms are not applicable to learning problems involving sampling

over ODE trajectories. For example, they are not suitable for learning approaches rooted in variational inference methods. An intriguing approach for accelerating the adjoint method is introduced by Daulbaev et al. (2020). These authors advocate the use of barycentric Lagrange interpolation (BLI) on activation values over a Chebyshev grid as an alternative to executing the backward pass over ODEs involving activation state variables. This approach presents a trade-off similar to ours, involving considerations of stability, accuracy and memory consumption. For instance, achieving an accurate approximation of ODE dynamics may necessitate employing a smaller time step, which in turn implies increased memory resource utilization. In contrast, our approach utilizes local approximations of the ODE solution, specifically a sequence of polynomials that must satisfy the ODE solution at consecutive time points. The BLI approach bears some resemblance to the use of global representations for ODE/PDE solutions using neural networks, as seen in Han et al. (2018). However, one notable advantage of the BLI approach is its theoretical foundation. Our approach aligns more closely with model predictive control numerical approximations, as described in Garcia et al. (1989). This methodology employs collocation methods to approximate ODEs, forming a nonlinear program. Furthermore, we provide justification for our block coordinate gradient descent algorithms by emulating a backpropagation algorithm based on a direct method for sensitivity analysis. There is previous work on carrying differentiation operators over the steps of ODE solvers. Farrell et al. (2013) applied adjoint methods to both ODEs and PDEs together with backpropagation over the forward steps of the ODE/PDE solvers, resulting in the `dolfin` library. Gradient estimations using direct sensitivity analysis methods was demonstrated by Carpenter et al. (2015). Choosing the direct or adjoint method depends on the problem. When the number of optimization variables dominates the size of the state vector, the adjoint methods are preferred. In contrast, when the number of optimization variables is small compared to the size of the state vector, direct methods are numerically more efficient. This was clear in the Cucker-Smale example, where the number of parameters remained constant, while the state vector dimension was increasing. Han et al. (2018) use global parameterization of solutions, as an alternative to direct collocation methods. They use neural networks to parameterize solutions of PDEs, where automatic differentiation is used the construct a loss function in terms of spatial and temporal differential operators. Boyd (2001) shows how spectral methods can be used to represent differential equations solutions as expansion of basis functions (e.g., Chebyshev, Fourier). Closer to our idea is the result introduced by Roesch et al. (2021), where the authors avoid using ODE solvers by approximating the state derivative using a collocation method. In their approach, the loss function is defined in terms of the approximation of the observed state derivative and the one predicted by a neural-ODE. The state needs to be fully observed. Our approach applies to partially observed systems, as well. In addition, we improve the speed of convergence via a coordinate descent approach.

5. Conclusions

ODE solvers endowed with sensitivity analysis capabilities have uses in many engineering applications (e.g., design, control or diagnosis) that employ model-based approaches. The integration of such solvers in DL platforms enables compositions with various neural network layers, and end to end support of automatic differentiation. Such solvers though

become slower as the complexity of the models increases. We presented two gradient descent algorithms based on block coordinate descent. They were designed by reformulating an optimization problem with dynamical constraint into an equivalent optimization problem with explicit equality constraints. The equality constraints are residual functions that reflect how close the optimization variables representing the estimate of an ODE solution are from the ODE solution. The residual functions were based on direct collocation methods to enable their parallel evaluation. The algorithms were implemented in the Pytorch and Jax frameworks and tested on the Cucker-Smale model of various complexities. In both frameworks the proposed algorithm are significantly faster and are at least as accurate as the gradient descent algorithms that use ODE solvers featuring sensitivity analysis. In addition, we tested the algorithms on learning black-box dynamical models, and showed how our algorithms can be used in conjunction with sensitivity-enabled ODE-based implementations, to speedup the training process.

References

Dimitri P. Bertsekas. *Nonlinear Programming*. Athena Scientific, 1999.

John P. Boyd. *Chebyshev and Fourier Spectral Methods*. Dover Books on Mathematics. Dover Publications, Mineola, NY, second edition, 2001.

James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, and Skye Wanderman-Milne. JAX: composable transformations of Python+NumPy programs, 2018. URL `http://github.com/google/jax`.

Yang Cao, Shengtai Li, and Linda Petzold. Adjoint sensitivity analysis for differential-algebraic equations: algorithms and software. *Journal of Computational and Applied Mathematics*, 149(1):171–191, 2002. Scientific and Engineering Computations for the 21st Century - Methodologies and Applications Proceedings of the 15th Toyota Conference.

Bob Carpenter, Matthew D. Hoffman, Marcus A. Brubaker, Daniel D. Lee, Peter Li, and Michael Betancourt. The Stan math library: Reverse-mode automatic differentiation in C++. *ArXiv*, abs/1509.07164, 2015.

Jose A. Carrillo, Massimo Fornasier, Jesus Rosado, and Giuseppe Toscani. Asymptotic flocking dynamics for the kinetic Cucker–Smale model. *SIAM Journal on Mathematical Analysis*, 42(1):218–236, 2010.

Ricky T. Q. Chen, Yulia Rubanova, Jesse Bettencourt, and David K Duvenaud. Neural ordinary differential equations. In *Neural Information Processing Systems*, 2018.

Talgat Daulbaev, Alexandr Katrutsa, Larisa Markeeva, Julia Gusak, Andrzej Cichocki, and Ivan Oseledets. Interpolation technique to speed up gradients propagation in neural ODEs. *ArXiv*, abs/2003.05271, 2020.

Robert P. Dickinson and Robert J. Gelinas. Sensitivity analysis of ordinary differential equation systems—a direct method. *Journal of Computational Physics*, 21(2):123–143, 1976.

Patrick E. Farrell, David A. Ham, Simon Wolfgang Funke, and Marie E. Rognes. Automated derivation of the adjoint of high-level transient finite element programs. *ArXiv*, abs/1204.5577, 2013.

Carlos E. Garcia, David M. Prett, and Manfred Morari. Model predictive control: theory and practice - a survey. *Automatica*, 25(3):335 – 348, 1989.

David J. Gardner, Daniel R. Reynolds, Carol S. Woodward, and Cody J. Balos. Enabling new flexibility in the SUNDIALS suite of nonlinear and differential/algebraic equation solvers. *ACM Transactions on Mathematical Software (TOMS)*, 2022.

Will Grathwohl, Ricky T. Q. Chen, Jesse Bettencourt, Ilya Sutskever, and David Duvenaud. FFJORD: free-form continuous dynamics for scalable reversible generative models. *International Conference on Learning Representations*, 2019.

Ernst Hairer and Gerhard Wanner. Stiff differential equations solved by Radau methods. *Journal of Computational and Applied Mathematics*, 111(1):93–111, 1999.

Jiequn Han, Arnulf Jentzen, and Weinan E. Solving high-dimensional partial differential equations using deep learning. *Proceedings of the National Academy of Sciences*, 115(34): 8505–8510, 2018.

Charles R. Hargraves and Stephen W. Paris. Direct trajectory optimization using nonlinear programming and collocation. *Journal of Guidance, Control, and Dynamics*, 10(4):338–342, 1987.

Albert L. Herman and Bruce A. Conway. Direct optimization using collocation based on high-order Gauss-Lobatto quadrature rules. *Journal of Guidance, Control, and Dynamics*, 19(3):592–599, 1996.

Matteo Hessel, David Budden, Fabio Viola, Mihaela Rosca, Eren Sezener, and Tom Hennigan. Optax: composable gradient transformation and optimisation, in JAX, 2020. URL `http://github.com/deepmind/optax`.

Magnus R. Hestenes. Multiplier and gradient methods. *Journal of Optimization Theory and Applications*, 4:303–320, 1969.

Alan C. Hindmarsh, Peter N. Brown, Keith E. Grant, Steven L. Lee, Radu Serban, Dan E. Shumaker, and Carol S. Woodward. SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers. *ACM Transactions on Mathematical Software (TOMS)*, 31(3):363–396, 2005.

Patrick Kidger. *On Neural Differential Equations*. PhD thesis, University of Oxford, 2021.

Patrick Kidger, Ricky T. Q. Chen, and Terry Lyons. "Hey, that's not an ODE": faster ODE adjoints via seminorms. *ArXiv*, abs/2009.09457, 2021.

Ernst Leonard Lindelöf. Sur l'application de la méthode des approximations successives aux équations différentielles ordinaires du premier ordre. *Comptes Rendus Hebdomadaires des Séances de l'Académie des Sciences*, pages 454–457, 1894.

Zoya Meleshkova, Sergei Evgenievich Ivanov, and Lubov Ivanova. Application of neural ODE with embedded hybrid method for robotic manipulator control. *Procedia Computer Science*, 193:314–324, 2021.

Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in Pytorch. In *NIPS 2017 Workshop Autodiff*, 2017.

Dimitris C. Psichogios and Lyle H. Ungar. A hybrid neural network-first principles approach to process modeling. *AIChE Journal*, 38(10):1499–1511, 1992.

Sashank J. Reddi, Satyen Kale, and Sanjiv Kumar. On the convergence of Adam and beyond. In *International Conference on Learning Representations*, 2018.

Elisabeth Roesch, Chris Rackauckas, and Michael Stumpf. Collocation based training of neural ordinary differential equations. *Statistical Applications in Genetics and Molecular Biology*, 20, 07 2021.

Yulia Rubanova, Ricky T. Q. Chen, and David K Duvenaud. Latent ordinary differential equations for irregularly-sampled time series. In *Neural Information Processing Systems*, 2019.

Juntang Zhuang, Nicha Dvornek, Xiaoxiao Li, Sekhar Tatikonda, Xenophon Papademetris, and James Duncan. Adaptive checkpoint adjoint method for gradient estimation in neural ODE. *ArXiv*, abs/2006.02493, 2020.