Progressive Batching L-BFGS Method for Machine Learning: Implementation and Comparison in Julia

Rohit Panse

Advisor: Brendan Keith
Reader: Peter Sentz

A Thesis submitted in partial fulfillment of the requirements for Honors
in the Departments of Applied Mathematics and Computer Science at Brown University

Department of Applied Mathematics
Brown University
Providence, Rhode Island

Abstract

Machine learning (ML) approaches are fundamentally linked to optimization problems, as most ML algorithms aim to minimize a loss function during the training process. Consequently, the efficiency of optimization techniques is crucial to the training and performance of ML models. While quasi-Newton (QN) methods demonstrate relatively faster convergence compared to other optimization techniques, they are primarily tailored for full-batch optimization and are thus less suitable for batching scenarios found in machine learning. As a result, QN methods are computationally expensive in relation to many first-order methods. A new L-BFGS method has been developed which combines progressive batching, a stochastic line search, and stable QN updating. This thesis implements this method in Julia and compares its efficiency to various first-order stochastic optimization methods as well as other QN methods. Algorithms were compared in terms of gradient evaluations using experiments on various neural ordinary differential equations, including a model for a quantum master equation. The progressive batching L-BFGS method emerged as the fastest algorithm, significantly outperforming first-order methods and other QN methods. This work showcases the potential of the progressive batching L-BFGS method, opening new avenues for further exploration and enhancement of QN methods.

Acknowledgments

I extend a sincere thank you to Professor Brendan Keith and Peter Sentz, whose mentorship over the past year and a half has facilitated an incredibly rewarding research journey. I could not be more grateful for their time, insights, kindness, and guidance. I am also extremely grateful to my parents for their unwavering support over the course of my academic journey. Finally, I would like to thank my sister, Shruti, and friend Shreyas, for their constant motivation.

Contents

1 Introduction

Machine learning (ML) has become ubiquitous, with applications spanning from the hard sciences to consumer-facing products. Optimization lies at the heart of ML, enabling models to learn optimal parameters of an objective function based on training data. With widespread adoption of increasingly sophisticated models and the tremendous complexity and size of datasets, the importance of optimization algorithms used to train these machine learning models cannot be understated. There is an increasing need for optimization algorithms that combine speed with computational and memory efficiency.

Among the models that benefit from such optimization techniques, neural ordinary differential equations (neural ODEs) are a family of deep neural network models. Rather than defining a discrete sequence of hidden layers, the hidden state continuously evolves via ODEs. The network's output can be computed via a black-box differential equation solver. Neural ODEs have constant memory cost, can trade numerical precision for speed, and adapt their evaluation strategy to each input [1].

Data-driven discovery is an experimental paradigm that tremendously benefits from the application of neural odes. Traditional theoretical methods for learning about the behavior underlying a dynamic system require explicit knowledge of the underlying differential equations governing the system. However, neural ODEs enable the modeling of a dynamical system by representing the derivative of the system's state with a neural network. Data discovery with neural ODEs present a unique optimization challenge due to the non-trivial computational cost required to compute gradients of the ODE solver, compounded by the need to handle large datasets effectively.

Optimization algorithms encompass a diverse range of methods, with first-order and second-order methods representing two important categories. First-order methods rely only on gradient information in order to update the parameters of the model. Second-order methods, on the other hand, update model parameters by incorporating both gradient and Hessian information, which provides an understanding of the curvature of the objective function [2]. Similarly, quasi-Newton (QN) methods, an offshoot of second-order methods, construct an approximation of the Hessian with gradient information in order to optimize parameters [3]. Since QN methods rely on high-quality gradients to estimate the curvature of the objective function and conduct effective line searches, they require large batch sizes. This results in a higher cost per iteration compared to first-order methods [4].

Bollapragada et al. proposed the progressive batching L-BFGS method, in which the sample size is initially small and increases as the algorithm progresses [5]. The progressive batching L-BFGS is appealing as it inherits the efficient initial behavior or first-order

methods, while incorporating second-order information, in addition to enabling the benefits of parallelism [6]. Additionally, it leverages the speed of QN methods while preserving the ability to progressively batch, features particularly useful in optimizing data discovery with neural ODE paradigms.

For this thesis, progressive batching L-BFGS is implemented in Julia and is compared to several first-order methods, as well as other QN methods. The evaluation is conducted on an important class of problems: data-driven discovery with neural ODEs. This thesis demonstrates progressive batching L-BFGS's computational efficiency and effectiveness in solving complex optimization problems, particularly in the context of neural ODEs.

1.1 Literature Review

Optimization algorithms that progressively increase batch size have been studied in both first order and second order contexts. Friedlander and Schmidt studied the finite sum problem and show increasing the sample size at a geometric rate yields linear convergence [7]. Byrd et al. examined the empirical risk minimization problem and propose a strategy based on the *norm* test, finding that when the sample size grows geometrically linear convergence is achieved [6]. Bollapragada et al. proposed the *inner product* test, improving upon the *norm* test, which similarly is used to determine the size of the batch [8].

There has been growing focus on understanding the generalization properties of progressive batching methods for training neural networks. Smith et al. demonstrated that decaying the learning rate during training can achieve the same learning curve as increasing the batch size. Both approaches have test accuracies that match default methods, but because fewer parameter updates are performed, which shortens training time and offers better parallelization opportunities [9]. Deverakonda et al. parallelized this approach [10]. De et al. displayed numerical results for the *norm test*, which was shown to have similar convergence rates to stochastic gradient descent [11]. Balles et al. presented a progressive batching scheme which combines the step-length with sample size [12].

Many methods have been proposed for data-driven discovery of dynamical systems. Crutchfield et al. described equation free modeling where the deterministic portion of equations of motion were reconstructed directly from a data series [13]. Schmidt et al. and Bongard et al. leveraged an evolutionary algorithm and symbolic regression in order to directly determine a non-linear dynamic system from data [14] [15]. Gonzales et al. demonstrated an artificial neural network approach, which was motivated by numerical discretization techniques used to solve partial differential equations [16]. Neural Ordinary Differential Equations (ODEs) as proposed by [1], present a promising framework for modeling dynamical systems, employing residual neural networks as approximate ODE solvers.

2 Optimization Algorithms

We consider supervised learning problems characterized by training data

$$T = \{(\hat{\boldsymbol{x}}_1, \hat{\boldsymbol{y}}_1), (\hat{\boldsymbol{x}}_2, \hat{\boldsymbol{y}}_2), \ldots, (\hat{\boldsymbol{x}}_n, \hat{\boldsymbol{y}}_n)\},$$

consisting of n input-output pairs, where the feature $\hat{\boldsymbol{x}}_i \in \mathbb{R}^{d_x}$ and $\hat{\boldsymbol{y}}_i \in \mathbb{R}^{d_y}$ are of dimensions d_x and d_y respectively. We define our model F as a mapping $F(\boldsymbol{\theta}; .) : \mathcal{X} \to \mathcal{Y}$, where $\mathcal{X}$ and $\mathcal{Y}$ denote the feature and the label space, respectively. F has d parameters given by the vector $\boldsymbol{\theta} \in \mathbb{R}^d$.

We can represent the error introduced by the model with loss $\ell(\boldsymbol{\theta}; \hat{\boldsymbol{x}}_i, \hat{\boldsymbol{y}}_i)$. Throughout this paper, error and loss will be used interchangeably. Various loss functions exist to model error, but in this study, we will focus on mean square error (MSE). The MSE can be used to measure how close the output of our model, $F(\boldsymbol{\theta}; \hat{\boldsymbol{x}}_i)$ is to the i-th data point, $\hat{\boldsymbol{y}}_i$. For example, $F(\boldsymbol{\theta}; \hat{\boldsymbol{x}}_i) = \hat{\boldsymbol{x}}_i^T \boldsymbol{\theta}$ is a linear model.

$$\ell(\boldsymbol{\theta}; \hat{\boldsymbol{x}}_i, \hat{\boldsymbol{y}}_i) = \frac{1}{n} \sum_{i=1}^{n} (F(\boldsymbol{\theta}, \hat{\boldsymbol{x}}_i) - \hat{\boldsymbol{y}}_i)^2 = \frac{1}{n} \sum_{i=1}^{n} (\hat{\boldsymbol{x}}_i^T \boldsymbol{\theta} - \hat{\boldsymbol{y}}_i)^2 \tag{1}$$

Given this MSE loss function, we can define a smooth unconstrained optimization problem as follows:

$$\boldsymbol{\theta}_* \in \arg\min_{\boldsymbol{\theta} \in \mathbb{R}^d} \frac{1}{n} \sum_{i=1}^{n} (\hat{\boldsymbol{x}}_i^T \boldsymbol{\theta} - \hat{\boldsymbol{y}}_i)^2 \tag{2}$$

Optimizing the MSE loss consists of finding the weights $\boldsymbol{\theta}$ that minimize the variation between the predicted values, $\hat{\boldsymbol{x}}_i^T \boldsymbol{\theta}$, and the actual values, $\hat{\boldsymbol{y}}_i$, over all of the data points $i = 1, 2, ..., n$.

With our optimization problem clearly defined, we will now explore different optimization methods to find the vector, $\boldsymbol{\theta}$, that minimizes MSE. In the following subsections, we will introduce the *Gradient Descent*, *ADAM*, *BFGS*, *L-BFGS*, *Progressive Batching L-BFGS* algorithms.

2.1 Gradient Descent

First order methods are one of the most widely used classes of optimization algorithms and has widespread use in training neural networks. One prominent example is the gradient descent algorithm [17]. Starting at an initial point $\boldsymbol{\theta}_0$, the algorithm constructs a sequence with the formula:

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \alpha \boldsymbol{\nabla} \ell(\boldsymbol{\theta}_k),$$

where α is the learning rate which decides the size of the step in the direction of the negative gradient. The sequence iteratively steps until a minimum is found. The gradient descent algorithm is shown in Algorithm

Algorithm 1 Gradient Descent algorithm

Require: $\ell(\boldsymbol{\theta})$: the objective function with parameters $\boldsymbol{\theta}$

Require: α : step size

Require: $\boldsymbol{\theta}_0$: initial parameters

1: **while** convergence is not reached **do**

2: $\quad \boldsymbol{g}_k \leftarrow \boldsymbol{\nabla}_{\boldsymbol{\theta}} \ell_k(\boldsymbol{\theta}_{k-1})$

3: $\quad \boldsymbol{\theta}_k \leftarrow \boldsymbol{\theta}_{k-1} - \alpha \boldsymbol{g}_k$

4: **end while**

This technique achieves linear convergence under sufficient regularity assumptions; that is, when the initial estimate $\boldsymbol{\theta}_0$ is sufficiently close to the optimum and when the step size α is sufficiently modest [18].

Each iteration of the full batch gradient descent involves visiting all the training samples in order to compute the average gradient. When n is large, training the model can be very slow and computationally expensive.

2.2 Gradient Descent with Mini-Batching

Gradient descent with mini-batching is a variation of the gradient descent algorithm that computes an estimate of the gradient using a random sub-sample of the data, known as a mini-batch. This mini-batch gradient estimate is used as a search direction in updating the parameters. Starting at the initial point $\boldsymbol{\theta}_0$, the algorithm constructs a sequence with the formula:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \alpha \boldsymbol{\nabla} \ell(\boldsymbol{\theta}_k; \boldsymbol{x}_{(i:i+n)}, \boldsymbol{y}_{(i:i+n)})$$

The gradient descent with mini-batching algorithm is shown in Algorithm 2.

Algorithm 2 Gradient Descent with Mini-Batching algorithm

Require: $\ell(\boldsymbol{\theta})$: the objective function with parameters $\boldsymbol{\theta}$

Require: α : step size

Require: b: size of full batch, n: mini-batch size

Require: $\boldsymbol{\theta}_0$: initial parameters

1: Initialize $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta}_0$
2: **while** convergence is not reached **do**
3: **for** $i = 1$ to $\lceil \frac{b}{n} \rceil$ **do**
4: $g_k \leftarrow \nabla \ell(\boldsymbol{\theta}_k; \boldsymbol{x}_{(i:i+n)}, \boldsymbol{y}_{(i:i+n)})$
5: $\boldsymbol{\theta}_k \leftarrow \boldsymbol{\theta}_{k-1} - \alpha g_k$
6: **end for**
7: **end while**

Mini-batching enables quick, cheap updates, which have relatively low variance compared to stochastic gradient methods. In addition, batching enables parallelization, which can further reduce training time.

Any optimization method of that involves the calculation of a gradient can employ a mini-batching approach to leverage the aforementioned benefits.

2.3 Adam

Adaptive moment estimation (Adam) is an optimization algorithm that is an adaptive learning rate method, meaning that the learning rate is automatically adjusted in order to achieve faster convergence [19].

In order to properly understand Adam, it is useful to examine algorithms that form its component parts: Heavy Ball Method and RMSprop.

2.3.1 Heavy Ball Method

The heavy ball method accelerates gradient descent by using a velocity vector, which is an accumulation of previous search directions over multiple iterations [20]. In regions of high curvature the velocity vector reduces oscillations, penalizing changes in directions which lead to smaller steps. On the other hand, in regions of low curvature the velocity vector accumulates momentum, leading to larger steps which accelerate convergence.

Intuitively, gradient descent is analogous to a man descending down a hill, taking the steepest path down, moving slow and steady. Gradient descent with the heavy ball method, however, can be described as a heavy ball rolling down the same hill. The tremendous inertia of the ball accelerates the path and smooths it over. The ball swiftly traverses tiny valleys, minor bumps, and local minima [21]. The heavy ball method has been shown to accelerate convergence.

Given the objective function $\ell(\boldsymbol{\theta})$, the following equations are used to update the pa-

rameters:

$$\boldsymbol{v}_k = \beta \boldsymbol{v}_{k-1} - (1-\beta)\boldsymbol{\nabla}\ell(\boldsymbol{\theta}_k)$$

$$\boldsymbol{\theta_k} = \boldsymbol{\theta}_{k-1} + \alpha \boldsymbol{v}_k$$

The parameter α is the learning rate, and β is recommended to be close to 0.9. The term $\boldsymbol{v}_k$ computes the exponentially decaying moving average of the gradient values. $\boldsymbol{v}_k$ tells the algorithm about the trends in the past gradient values; this heavy ball term decreases updates for dimensions whose gradients change directions and increases updates for dimensions whose gradients point in the same direction [22].

Figure 1 illustrates what these updates look like and how the heavy ball method accelerates the training process.

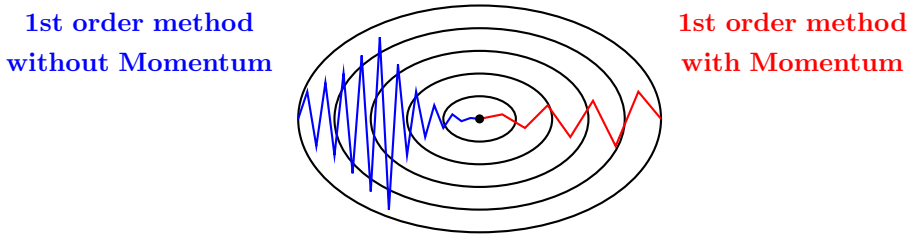

Figure 1: Trajectory of 1st order method with and without momentum. Based on Figure 1 in [23].

2.3.2 RMSProp

RMSProp, which stands for root mean square propagation, uses adaptive learning rates to update parameters [24]. The algorithm maintains a decaying moving average of the squares of the gradients [25]. Moving averages help normalize gradient updates, leading to the stabilization of the learning process. When updating the parameters, the learning rate is divided by the square root of the moving average. The update looks like the following:

$$\boldsymbol{s}_k = \beta \boldsymbol{s}_{k-1} + (1-\beta)\boldsymbol{\nabla}\ell(\boldsymbol{\theta}_k)^2$$

$$\boldsymbol{\theta}_k = \boldsymbol{\theta}_k - \frac{\alpha}{\sqrt{\boldsymbol{s}_k + \epsilon}}\boldsymbol{\nabla}\ell(\boldsymbol{\theta}_k)^2$$

The parameter α is the learning rate, and β is recommended to be close to 1. The adaptive learning rate strategy of RMSProp enables the algorithm to prioritize the most recent shape of the search space, leading to faster convergence.

2.3.3 Putting it together: Adam

The main idea of the Adam algorithm centers on tracking two different moving averages: m_t and v_t [19]. These are represented formulaically below:

$$\boldsymbol{m}_k = \beta_1 \boldsymbol{m}_{k-1} + (1-\beta_1)\boldsymbol{\nabla}\ell(\boldsymbol{\theta}_k)$$
$$\boldsymbol{v}_k = \beta_2 \boldsymbol{v}_{k-1} + (1-\beta_2)\boldsymbol{\nabla}\ell(\boldsymbol{\theta}_k)^2$$

The variable $\boldsymbol{m}_k$ represents an exponentially decaying average of previous gradients, similar to the quantity tracked in the heavy ball method algorithm. The variable $\boldsymbol{v}_k$ represents an exponentially decaying average of previous squared gradients, similar to the quantity tracked in the RMSProp algorithm.

In order to update the parameters for a particular epoch, the algorithm calculates the intermediary values, $\hat{\boldsymbol{m}}_k$ and $\hat{\boldsymbol{v}}_k$, represented by the forumulae below:

$$\hat{\boldsymbol{m}}_k = (\frac{\boldsymbol{m}_k}{1-\beta_1^k}) \quad \hat{\boldsymbol{v}}_k = (\frac{\boldsymbol{v}_k}{1-\beta_2^k}).$$

The terms $\hat{\boldsymbol{m}}_k$ and $\hat{\boldsymbol{v}}_k$ are designed to correct the biases toward zero in $\boldsymbol{m}_k$ and $\boldsymbol{v}_k$, which can occur during the initial time steps or when the decay rates are small (i.e. β_1 and β_2 are nearly 1).

Finally, these intermediary terms are used to update the weights in a manner similar to RMSProp and the heavy ball method, as follows:

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \frac{\eta}{\sqrt{\hat{\boldsymbol{v}}_k + \epsilon}}\hat{\boldsymbol{m}}_k$$

The learning rate is represented by η. The default values recommended for the parameters are $\beta_1 = 0.9$, $\beta_2 = 0.999$, and $\epsilon = 10^{-8}$. The Adam algorithm is shown in Algorithm 3.

Algorithm 3 Adam algorithm

Require: $\ell(\boldsymbol{\theta})$: the objective function with parameters $\boldsymbol{\theta}$
Require: α : step size
Require: β_1, β_2: exponential decay rates
Require: $\boldsymbol{\theta}_0$: initial parameters
1: $\boldsymbol{m}_0 \leftarrow \boldsymbol{0}, \boldsymbol{v}_0 \leftarrow \boldsymbol{0}, k = 0$
2: **while** convergence is not reached **do**
3: $k \leftarrow k + 1$
4: $\boldsymbol{g}_k \leftarrow \nabla_{\boldsymbol{\theta}}\ell_k(\boldsymbol{\theta}_{k-1})$
5: $\boldsymbol{m}_k \leftarrow \beta_1 \boldsymbol{m}_{k-1} + (1-\beta_1)\boldsymbol{g}_k$

6: $\boldsymbol{v}_k \leftarrow \beta_2 \boldsymbol{v}_{k-1} + (1-\beta_2)\boldsymbol{\nabla} \boldsymbol{g}_k^2$
7: $\hat{\boldsymbol{m}}_t = \boldsymbol{m}_k/(1-\beta_1^k),\ \hat{\boldsymbol{v}}_k = \boldsymbol{v}_k/(1-\beta_2^k)$
8: $\boldsymbol{\theta}_k \leftarrow \boldsymbol{\theta}_{k-1} - \alpha(\hat{\boldsymbol{m}}_k \sqrt{\hat{\boldsymbol{v}}_k} + \epsilon)$
9: **end while**

2.4 Quasi-Newton Methods

2.4.1 Comparison to Newton Methods

Newton methods are a class of optimization algorithms that utilize second-order information about the objective function, specifically in the form of the Hessian matrix. The Hessian is used to determine search directions that effectively incorporate the function's curvature. The step direction $\boldsymbol{d}_k$ is calculated as follows.

$$\boldsymbol{G}_k \boldsymbol{d}_k = -\boldsymbol{g}_k \tag{3}$$

where $\boldsymbol{G}_k \equiv \nabla^2 f(\boldsymbol{x}_k)$. If f is twice continuously differentiable and the Hessian matrix is positive definite, the convergence order is Q-quadratic, indicating a relatively rapid convergence. Specifically, the error in each iteration decreases proportionally to the square of the previous iteration [2]. However, achieving this speed of convergence has trade-offs. Namely, the computation of the Hessian can be very computationally expensive, taking up to $O(n^3)$ operations. Additionally, maintaining the positive definite property of the Hessian may not be possible with nonlinear and high-dimensional functions.

Quasi-Newton methods, on the other hand, remedy some of the computational inefficiencies of Newton methods. The general form of the quasi-Newton method calculates $\boldsymbol{p}_k$ with the following function:

$$\boldsymbol{B}_k \boldsymbol{p}_k = -\boldsymbol{g}_k \tag{4}$$

where $\boldsymbol{B}_k \in \mathbb{R}^{n \times n}$ is an approximation of the Hessian matrix $\boldsymbol{G}_k$ [26], which is typically updated based on gradient evaluations at each iteration. In doing so, quasi-Newton methods provide the benefit of improving the convergence rate, without the expensive computational costs of Newton methods.

2.4.2 Line Search: Wolfe conditions

Inexact line search strategies involve calculating the step length α that achieves a sufficient decrease in ℓ at minimal cost. The Wolfe conditions are a popular line search algorithm

in which a sequence of candidate values for α are tested until the Armijo and curvature conditions are satisfied [27].

The Armijo condition ensures sufficient decrease in ℓ by imposing an inequality ensuring that the reduction in ℓ is proportional to both the step length α and the directional derivative $\nabla \ell_k^T \boldsymbol{p}_k$

$$\ell(\boldsymbol{\theta_k} + \alpha \boldsymbol{p}_k) < \ell(\boldsymbol{\theta}_k) + c_1 \alpha \nabla \ell_k^T \boldsymbol{p}_k, \tag{5}$$

for some constant $c_1 \in (0, 1)$. To prevent the selection of unacceptably small α's which satisfy Equation (5), the curvature condition is introduced to ensure reasonable progress is made. This requires α to satisfy

$$\nabla \ell(\boldsymbol{\theta}_k + \alpha \boldsymbol{p}_k)^T \boldsymbol{p}_k \geq c_2 \nabla \ell_k^T \boldsymbol{p}_k,$$

for some constant $c_2 \in (c1, 1)$ The curvature condition ensures that the selected α enables an appropriate reduction of the gradient $\nabla \ell$.

Wolfe conditions can be used in most line-search methods and have particular importance in the implementation of QN methods.

2.4.3 BFGS

The Broyden-Fletcher-Goldfarb-Shanno (BFGS) method is a quasi-Newton optimization technique that leverages curvature information through an iterative approximation of the Hessian matrix [28, 29, 30, 31].

The BFGS method relies on the construction of a quadratic model of the objective function at the current iterate $\boldsymbol{\theta}_k$.

$$\boldsymbol{m}_k(\boldsymbol{p}) = f(\boldsymbol{\theta}_k) + \nabla f(\boldsymbol{\theta}_k)^T \boldsymbol{p} + \frac{1}{2} p^T \boldsymbol{B}_k \boldsymbol{p} \tag{6}$$

where $\boldsymbol{B}_k$ is a positive definite matrix representing the approximation of the Hessian matrix.

The minimizer, $\boldsymbol{p}_k$, of the convex quadratic model is calculated as follows:

$$\boldsymbol{p}_k = -\boldsymbol{B}_t^{-1} \nabla f(\boldsymbol{\theta}_k). \tag{7}$$

Consequently, $\boldsymbol{p}_k$ is used as a search direction when calculating the value the subsequent iterate:

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k + \alpha_k \boldsymbol{p}_k, \tag{8}$$

where the step length a_k is selected to satisfy the Wolfe conditions.

To refine the Hessian approximation $\boldsymbol{B}_k$, BFGS uses the difference in gradients between consecutive iterations. Given vectors $\boldsymbol{s}_k = \boldsymbol{\theta}_{k+1} - \boldsymbol{\theta}_k$ and $\boldsymbol{y}_k = \nabla f(\boldsymbol{\theta}_{k+1}) - \nabla f(\boldsymbol{\theta}_k)$, the next Hessian approximation $\boldsymbol{B}_{k+1}$ is obtained through the update:

$$\boldsymbol{B}_{k+1} = \boldsymbol{B}_k - \frac{\boldsymbol{B}_k \boldsymbol{s}_k \boldsymbol{s}_k^T \boldsymbol{B}_k}{\boldsymbol{s}_k^T \boldsymbol{B}_k \boldsymbol{s}_k} + \frac{\boldsymbol{y}_k \boldsymbol{y}_k^T}{\boldsymbol{y}_k^T \boldsymbol{s}_k}, \tag{9}$$

The BFGS algorithm is shown in Algorithm 4.

Algorithm 4 BFGS Method

Require: Initial point x_0, convergence tolerance ϵ, initial Hessian approximation H_0

$k \leftarrow 0$
while $\|\nabla f(\boldsymbol{\theta}_k)\| > \epsilon$ **do**
  Compute search direction $\boldsymbol{p}_k = -\boldsymbol{H}_k \nabla f(\boldsymbol{\theta}_k)$
  Determine step length α_t to satisfy Wolfe conditions
  Update $\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k + \alpha_k \boldsymbol{p}_k$
  Define $\boldsymbol{s}_k = \boldsymbol{\theta}_{k+1} - \boldsymbol{\theta}_k$ and $\boldsymbol{y}_k = \nabla f(\boldsymbol{\theta}_{k+1}) - \nabla f(\boldsymbol{\theta}_k)$
  Compute $\boldsymbol{\rho}_k = \frac{1}{\boldsymbol{y}_k^T \boldsymbol{s}_k}$
  Update $\boldsymbol{H}_{k+1} = (I - \boldsymbol{\rho}_k \boldsymbol{s}_k \boldsymbol{y}_k^T)^T \boldsymbol{H}_k (I - \boldsymbol{\rho}_k \boldsymbol{y}_k, \boldsymbol{s}_t^T) + \boldsymbol{\rho}_k \boldsymbol{s}_k \boldsymbol{s}_k^T$
  $k \leftarrow k + 1$
end while

2.4.4 L-BFGS

Limited memory BFGS, or L-BFGS, provides a remedy to the intractability of $\boldsymbol{B}_k$, which is an $n \times n$ matrix. Unlike BFGS, which stores the complete approximation to the inverse Hessian, L-BFGS stores a fixed number m of the latest $\boldsymbol{s}_k = \boldsymbol{\theta}_{k+1} - \boldsymbol{\theta}_k$ and $\boldsymbol{y}_k = \nabla f(\boldsymbol{\theta}_{k+1}) - \nabla f(\boldsymbol{\theta}_k)$ vectors [32].

L-BFGS requires $O(mn)$ operations per iteration, and is far better suited for problems with high dimensionality.

The two-loop recursion algorithm is shown in Algorithm 5, which is an instrumental component to the L-BFGS, shown in Algorithm 6.

Algorithm 5 Two-loop Recursion

Input: Gradient $\boldsymbol{g}_k$, curvature pairs $\{(\boldsymbol{s}_i, \boldsymbol{y}_i)\}$

Set $\boldsymbol{q} = -\boldsymbol{g}_k$
for $i = k - 1$ to $k - m$ **do**
  $\boldsymbol{\rho}_i := \frac{1}{\boldsymbol{y}_i^T \boldsymbol{s}_i}$
  $\alpha_i = \boldsymbol{\rho}_i \boldsymbol{s}_i^T \boldsymbol{q}$

$\boldsymbol{q} = \boldsymbol{q} - \alpha_i \boldsymbol{y}_i$
end for
$\boldsymbol{r} = \boldsymbol{H}_0^k \boldsymbol{q}$
for $i = k - m$ to $k - 1$ **do**
$\beta = \boldsymbol{\rho}_i \boldsymbol{y}_i^T \boldsymbol{r}$
$\boldsymbol{r} = \boldsymbol{r} + \boldsymbol{s}_i(\alpha_i - \beta)$
end for
Output: $r = -\boldsymbol{H}_k \nabla \boldsymbol{F}(\boldsymbol{\theta}_k)$

Algorithm 6 L-BFGS Algorithm

Input: Initial point $\boldsymbol{\theta}_0$, integer $m > 0$

Output: Optimized x

for $k = 1, 2, \ldots$ **do**
Choose $\boldsymbol{H}_0^k$
Compute direction $\boldsymbol{p}_k = -\boldsymbol{H}_k \nabla f(\boldsymbol{\theta}_k)$ using Algorithm 4
Choose a learning rate $\alpha_k > 0$
Update $\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k + \alpha_k \boldsymbol{p}_k$
Update curvature pairs:
$\boldsymbol{s}_k = \boldsymbol{\theta}_{k+1} - \boldsymbol{\theta}_k$, $\boldsymbol{y}_k = \nabla F(\boldsymbol{\theta}_{k+1}) - \nabla F(\boldsymbol{\theta}_k)$
if $k \geq m$ **then**
Replace the oldest pair $(\boldsymbol{s}_{k-m}, \boldsymbol{y}_{k-m})$ with $(\boldsymbol{s}_k, \boldsymbol{y}_k)$
else
Store the vector pair $(\boldsymbol{s}_k, \boldsymbol{y}_k)$
end if
end for

2.5 Progressive Batching L-BFGS

The Progressive Batching L-BFGS method creates adaptable batch sizes that start small and progressively increase in order to achieve a fast local rate of convergence and permit the use of second order information [5]. Whereas the L-BFGS relies on the whole data set for gradient calculations, leading to computational inefficiency, especially when the data-set is large. Progressive Batching L-BFGS balances computational efficiency with gradient accuracy, providing the benefits of QN updates without the high computational costs traditionally associated with L-BFGS.

2.5.1 Stochastic QN method

Stochastic QN updates calculate the next iterate of our optimization problem as follows:

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k + \alpha_k \boldsymbol{p}_k,$$

where the search direction $\boldsymbol{p}_k$ is calculated as

$$\boldsymbol{p}_k = \boldsymbol{H}_k \boldsymbol{g}_k^{S_k}$$

where $\boldsymbol{H}_k$ is a positive definite quasi-Newton matrix and the batch, or sub sampled gradient, is given by

$$\boldsymbol{g}_t^{S_k} = \nabla \ell_{S_k}(\boldsymbol{x}_k) = \frac{1}{|S_k|} \sum_{i \in S_k} \nabla \ell_i(\boldsymbol{x}_k),$$

with the set $S_k \subset \{1, 2, ...\}$ indexing the data points $(\hat{\boldsymbol{x}}_i, \hat{\boldsymbol{y}}_i)$.

2.5.2 Sample Size Selection

The inner product quasi-Newton (IPQN) test determines when, and by how much to increase the batch size $|S_k|$ over the course of the optimization procedure based on observed gradients. The test is calculated as follows

$$\frac{\mathrm{Var}_{i \in S_v^k}((\boldsymbol{g}_i^k)^T \boldsymbol{H}_k^2 \boldsymbol{g}_{S_k}^k)}{|S_k|} \leq \theta^2 \left\| \boldsymbol{H}_k \boldsymbol{g}_k^{S_k} \right\|^4, \tag{10}$$

for some $\theta > 0$, where $S_v^k \subseteq S_k$ is a subset of the current sample (batch), and the variance term is defined as

$$\mathrm{Var}_{i \in S_v^k}((\boldsymbol{g}_i^k)^T \boldsymbol{H}_k^2 \boldsymbol{g}_{S_k}^k) \equiv \frac{\sum_{i \in S_v^k} \left((\boldsymbol{g}_k^i)^T \boldsymbol{H}_k^2 \boldsymbol{g}_k^{S_k} - \| \boldsymbol{H}_k \boldsymbol{g}_k^{S_k} \|^2 \right)^2}{(|S_v^k| - 1)}.$$

Whenever equation (10) is not satisfied, we calculate an estimation of the increase in sample size $|S_k|$ required to satisfy the condition. We set the new sample size as $|S_{k+1}| = \lceil b_k \rceil$, where

$$b_k = \frac{\mathrm{Var}_{i \in S_v^k}((\boldsymbol{g}_i^k)^T \boldsymbol{H}_k^2 \boldsymbol{g}_{S_k}^k)}{\theta^2 \left\| \boldsymbol{H}_k \boldsymbol{g}_k^{S_k} \right\|^4}.$$

2.5.3 Line Search

The line search is a particularly important feature of the progressive batching L-BFGS algorithm because it ensures the robustness and efficiency of the iteration with little additional cost. The initial step length a_t is calculated as

$$a_t = \left(1 + \frac{\mathrm{Var}_{i \in S_v^k}((\boldsymbol{g}_i^k)^T \boldsymbol{H}_k^2 \boldsymbol{g}_{S_k}^k)}{|S_t| \left\| \boldsymbol{g}_k^{S_k} \right\|^2}\right)^{-1}$$

where $S_k^u \subseteq S_k$ and

$$\mathrm{Var}_{i \in S_v^k}\{\boldsymbol{g}_k^i\} = \frac{1}{|S_v^k| - 1} \sum_{i \in S_k^v} \left\| \boldsymbol{g}_k^i - \boldsymbol{g}_k^{S_k} \right\|^2$$

Starting with the initial step length a_t, a backtracking line search is performed to satisfy the Armijo condition as follows:

$$F^{S_k}(\boldsymbol{x}_k - \alpha_k \boldsymbol{H}_k \boldsymbol{g}_k^{S_k}) \leq F^{S_k}(\boldsymbol{x}_k) - c_1 \alpha_k (\boldsymbol{g}_k^{S_k})^T \boldsymbol{H}_k \boldsymbol{g}_k^{S_k}, \tag{11}$$

where $c_1 > 0$.

2.5.4 Stable Quasi-Newton Updates

Progressive batching L-BFGS updates the inverse Hessian in a manner very similar to BFGS and L-BFGS, with one difference. The update is given by

$$\begin{aligned} \boldsymbol{H}_{k+1} &= \boldsymbol{V}_k^T \boldsymbol{H}_k \boldsymbol{V}_k + \boldsymbol{\rho}_k \boldsymbol{s}_k \boldsymbol{s}_k^T, \\ \boldsymbol{\rho}_k &= (\boldsymbol{y}_k^T \boldsymbol{s}_k)^{-1}, \\ \boldsymbol{V}_k &= I - \boldsymbol{\rho}_k \boldsymbol{y}_k \boldsymbol{s}_k^T. \end{aligned} \tag{12}$$

where $\boldsymbol{s}_k = \boldsymbol{x}_{k+1} - \boldsymbol{x}_k$. The key difference lies in the definition of $\boldsymbol{y}_k$, which is computed as $\boldsymbol{y}_k = \boldsymbol{g}_{k+1}^{S_k} - \boldsymbol{g}_k^{S_k}$. For each batch S_k, the gradient is evaluated twice: once at $\boldsymbol{x}_k$ and once at $\boldsymbol{x}_{k+1}$; Because the batch changes from one iteration to the next ($S_{k+1} \neq S_k$), ensuring that a consistent S_k is used in the calculation of $\boldsymbol{y}_k$ is crucial to maintaining stability in the updating process.

The Progressive Batching L-BFGS is shown in Algorithm 7.

Algorithm 7 Progressive Batching L-BFGS Method

Input: Initial iterate θ_0, initial sample size $|S_0|$

Initialization: Set $k \leftarrow 0$

Repeat until convergence

1: Sample $S_k \subset \{1, \ldots, N\}$ with sample size $|S_k|$

2: **if** $\mathrm{Var}_{i \in S_k}\left(\frac{(\boldsymbol{g}_k^i)^T \boldsymbol{H}_k^2 \boldsymbol{g}_k^{S_k}}{|S_k|}\right) > \theta^2 \left\| \boldsymbol{H}_k \boldsymbol{g}_k^{S_k} \right\|^4$ **then**

3: Compute $b_k = \mathrm{Var}_{i \in S_k}\left(\frac{(\boldsymbol{g}_k^i)^T \boldsymbol{H}_k^2 \boldsymbol{g}_k^{S_k}}{|S_k|}\right)$

4: Compute $\hat{b}_k \leftarrow \lceil b_k \rceil - |S_k|$

Sample $S^+ \subset \{1, \ldots, N\} \setminus S_k$ with $|S^+| = \hat{b}_k$

Update $S_k \leftarrow S_k \cup S^+$

end if

Compute $\boldsymbol{g}_{S_k}$

Compute $\boldsymbol{p}_k = -\boldsymbol{H}_k \boldsymbol{g}_{S_k}^k$ using L-BFGS Two-Loop Recursion (Algorithm 4)

Compute $\alpha_k = \left(1 + \frac{\mathrm{Var}_{i \in S_k}\{\boldsymbol{g}_i^k\}}{|S_k|(\boldsymbol{g}_{S_k}^k)^2}\right)^{-1}$

while the Armijo condition, $F^{S_k}(\boldsymbol{x}_k - \alpha_k \boldsymbol{H}_k \boldsymbol{g}_k^{S_k}) \leq F^{S_k}(\boldsymbol{x}_k) - c_1 \alpha_k (\boldsymbol{g}_k^{S_k})^T \boldsymbol{H}_k \boldsymbol{g}_k^{S_k}$, is not satisfied **do**

Set $\alpha_k = \alpha_k / 2$

end while

Compute $\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k + \alpha_k \boldsymbol{p}_k$

Compute $\boldsymbol{y}_k = \boldsymbol{g}_{k+1}^{S_k} - \boldsymbol{g}_k^{S_k}$ or $\boldsymbol{y}_k = \boldsymbol{g}_{k+1}^{O_k} - \boldsymbol{g}_k^{O_k}$, where $O_k = S_k \cap S_{k+1}$

Compute $\boldsymbol{s}_k = \boldsymbol{\theta}_{k+1} - \boldsymbol{\theta}_k$

if $\boldsymbol{y}_k^T \boldsymbol{s}_k > \epsilon \|\boldsymbol{s}_k\|^2$ **then**

if number of stored $(\boldsymbol{y}_j, \boldsymbol{s}_j)$ exceeds m **then**

Discard the oldest curvature pair $(\boldsymbol{y}_j, \boldsymbol{s}_j)$

end if

Store the new curvature pair $(\boldsymbol{y}_k, \boldsymbol{s}_k)$

end if

Set $k \leftarrow k + 1$

Set $|S_k| = |S_{k-1}|$

3 Neural ODEs

Neural ODEs are a family of deep neural network models that are well suited to learning time dependent dynamical systems. They are continuous depth models which are equipped with the benefits of constant memory cost and the ability to explicitly trade numerical precision for speed [1].

In this work, we address the data driven discovery with neural ODEs framework. In this setup, we have time series data points

$$T = \{(t_1, \hat{\boldsymbol{y}}(t_1)), (t_2, \hat{\boldsymbol{y}}(t_2)), \ldots, (t_n, \hat{\boldsymbol{y}}(t_n))\},$$

consisting of n datapoints, with each pair $(t_i, \hat{\boldsymbol{y}}(t_i))$ corresponding to a time t_i and the observed data $\boldsymbol{y}(t_i)$ at that time.

Explicit knowledge of the underlying differential equations governing the system remains unknown. Thus, to model this setup, we define the following initial value problem (IVP):

$$\begin{cases} \frac{d\boldsymbol{y}(t)}{dt} = \boldsymbol{f}_\theta(t, \boldsymbol{y}(t)) \\ \boldsymbol{y}(0) = \boldsymbol{y}_0 \end{cases},$$

Where $\boldsymbol{y} : \mathbb{R} \to \mathbb{R}^d$, $d \in \mathbb{N}$, $t \in \mathbb{R}^+$, $\boldsymbol{f}_\theta : \mathbb{R} \times \mathbb{R}^d \to \mathbb{R}^d$ and $\boldsymbol{\theta} \in \mathbb{R}^p$, $p \in \mathbb{N}$. The function $\boldsymbol{f}_\theta$ is a neural network, where the input variable t corresponds to time points in our ODE. Given this setup, our goal is to find the most optimal neural network, $\boldsymbol{f}_\theta$, using the observed data points for $\boldsymbol{y}(t)$ over time.

Given a neural network $\boldsymbol{f}_\theta$, with parameters $\boldsymbol{\theta}$, that sufficiently models a system's dynamics, we can use an ODE solver to calculate the trajectories, $\boldsymbol{y}(t)$.

This neural ODE setup naturally leads us to an optimization problem, which is the main focus of this thesis. We define the loss using the MSE, as follows:

$$\text{MSE} = \frac{1}{n} \sum_{i=1}^{n} (\boldsymbol{y}(t_i) - \hat{\boldsymbol{y}}(t_i))^2$$

Where n represents the number of data points in the dataset, $\boldsymbol{y}(t_i)$ is value at the i-th time point predicted via the neural network, and $\hat{\boldsymbol{y}}(t_i)$ is the actual observed value at the i-th time point. The optimization problem involves minimizing the MSE, thereby enhancing the accuracy of the predictions from the neural ODE.

3.1 ODEs

ODEs are differential equations whose unknowns are functions of a single variable. An ordinary differential equation of the order n is defined by the relation

$$\boldsymbol{F}(t, \boldsymbol{y}, \boldsymbol{y}^{(1)}(t), \boldsymbol{y}^{(2)}(t), ..., \boldsymbol{y}^{(n)}(t)) = 0$$

where $\boldsymbol{y}^{(n)}$ stands for the n^{th} derivative of unknown function $t \mapsto \boldsymbol{y}(t)$ with respect to the independent variable t [33]. Ordinary differential equations (ODEs) are used to model and describe many processes in many fields including physics, engineering, chemistry, biology, and economics.

3.1.1 Explicit First-Order ODE

A first-order ODE can be defined by the following general relation:

$$\boldsymbol{F}(t, \boldsymbol{y}(t), \boldsymbol{y}'(t)) = 0$$

Where t is the independent variable, $\boldsymbol{y}(t)$ is the dependent variable, and $\boldsymbol{y}'(t)$ is the first derivative of y with respect to t. It is important to note that the independent variable is represented by t, which represents time. This is because dynamical systems often involve the state $\boldsymbol{y}(t)$ changing as t progresses. The general relation can be rewritten in explicit terms as the following IVP:

$$\begin{cases} \boldsymbol{y}'(t) = \boldsymbol{f}(t, \boldsymbol{y}) \\ \boldsymbol{y}(0) = \boldsymbol{y}_0 \end{cases}$$

Using the explicit form of the First-Order ODE is useful for clarity when calculating numerical solutions and will be the primary form used throughout this paper.

3.1.2 Euler's Method

Analytically computing solutions to ODEs can prove to be challenging and impractical, so numerical methods are often employed. Euler's method is the most elementary numerical method.

In order to calculate the solution to $\boldsymbol{y}'(t) = \boldsymbol{f}(t, \boldsymbol{y})$ passing through $(t_0, \boldsymbol{y}_0)$, we select a step size $\boldsymbol{h}$. Starting from $(t_0, \boldsymbol{y}_0)$, we approximate the curve over the interval $[t_0, t_0 + \boldsymbol{h}]$, using the tangent line with slope $\boldsymbol{f}(t_0, \boldsymbol{y}_0)$. The next point $(t_1, \boldsymbol{y}_1)$ is calculated with the equations:

$$t_1 = t_0 + \boldsymbol{h}, \qquad \boldsymbol{y}(t_1) = \boldsymbol{y}(t_0) + \boldsymbol{h} f(t_0, \boldsymbol{y}(t_0))$$

Now, we repeat the process to find the next points $(t_2, \boldsymbol{y}(t_2))$, using as the line segment having slope $\boldsymbol{f}(t_1, \boldsymbol{y}(t_1))$ as the new approximation to the curve. $(t_2, \boldsymbol{y}(t_2))$ are calculated by the equations

$$t_2 = t_1 + \boldsymbol{h}, \qquad t_2 = t_1 + \boldsymbol{h}\boldsymbol{f}(t_1, \boldsymbol{y}(t_1))$$

We continue constructing the approximation to the solution of the curve in the same way. The general formulas telling us how to get from the $(n-1)$ point to the n^{th} point are

$$t_n = t_{n-1} + \boldsymbol{h}, \qquad \boldsymbol{y}(t_n) = \boldsymbol{y}(t_{n-1}) + \boldsymbol{h}\boldsymbol{f}(t_{n-1}, \boldsymbol{y}(t_{n-1}))$$

In practice, we employ this formula until final time $T = t_{\text{final}}$ to construct an approximation of the curve consisting of line segments joining the points $((t_0, \boldsymbol{y}(t_0)), (t_1, \boldsymbol{y}(t_1)), \ldots, (t_{\text{final}}, \boldsymbol{y}(t_{\text{final}}))$ [34].

3.2 Artificial Neural Networks

Artificial neural networks are a biologically inspired programming paradigm that enables a computer to learn from observational data [35]. They consist of layers of computational units known as "neurons". Each neuron computes its value based on a linear combination of values of units that point into it and an activation function. Figure 2 illustrates a neural network, where each neuron computes its values based on the neurons connected to it via incoming arrows.

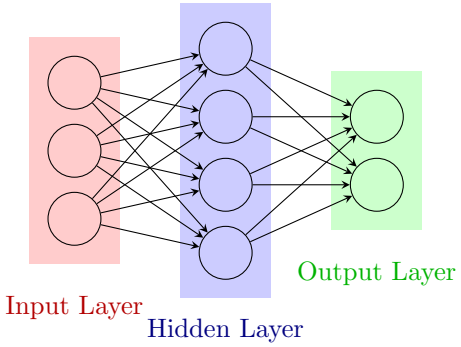

Figure 2: Visualizations of a 2-layer neural network. In this example, there are 3 input units, 4 hidden units, and 2 output units. Based on figure under the "Layer-wise organization" section of [36].

The training process for an ANN involves optimizing the weights and biases that are constituent components of each layer to minimize the loss. The structure of neural networks

allow for it to extract valuable insights from the data, learning patterns layer by layer. This makes them well suited for learning complex non-linear relationships.

3.3 Residual Networks

RNNs extend the capabilities of standard ANNs by incorporating connections that allow the network to maintain information across different steps, enabling it to effectively model time dependent data.

Residual networks build complicated transformations by composing a sequence of transformations to a hidden state [1]:

$$\boldsymbol{h}_{t+1} = \boldsymbol{h}_t + f(\boldsymbol{h}_t, \theta_t),$$

RNN's apply small, iterative updates to the input as it passes through the layers. Notably, the construction of this sequence has a similar form to Euler's method. Euler's method discretizes the continuous relationship between input and output domains. Likewise, Residual neural networks discretizes this continuous relationship via hidden states, $\boldsymbol{h}_t$. Similar to how an Euler's method update depends on the previous value, an RNN update to the hidden state directly depends on the previous hidden state.

3.4 Neural ODE

Upon taking a continuous limit of each discrete layer in the network (i.e. adding more layers with smaller steps), we parameterize the continuous dynamics of hidden states using an ordinary differential equation (ODE) specified by a neural network, also known as a neural ODE.

$$\frac{d\boldsymbol{h}(t)}{dt} = f(t, \boldsymbol{h}_t, \boldsymbol{\theta}_t)$$

Given an initial condition $\boldsymbol{y}(0)$, the output from the neural ODE, $\boldsymbol{y}(T)$, is defined as the solution to the ODE at time T.

$$\boldsymbol{y}(T) = \boldsymbol{y}(0) + \int_0^T \frac{d\boldsymbol{y}(T)}{dt}\, dt = \boldsymbol{y}(0) + \int_0^T \boldsymbol{f}(t, \boldsymbol{h}_t, \boldsymbol{\theta}_t) dt$$

This value can be computed using a black-box differential equation solver, which evaluates the hidden unit dynamics function $\boldsymbol{f}$ as needed to obtain the solution with the required accuracy.

3.4.1 Adjoint Method

Backpropogation is a method for training artificial neural networks, which adjusts weights and biases based on the error rate of predictions [37]. The neural ODE paper describes an algorithm for backpropogating through the continuous hidden state dynamics.

The loss function is defined as

$$L(\boldsymbol{h}(t_1)) = L(\text{ODESolve}(\boldsymbol{h}(t_0), G, t_0, t_1, \boldsymbol{\theta})).$$

In order to optimize L, gradients with respect to state $\boldsymbol{h}(t)$, time t, and parameters $\boldsymbol{\theta}$. The adjoint method describes a method to calculate this gradient. To keep track of time dynamics, the adjoint state is defined as

$$\boldsymbol{a}(t) = -\frac{\partial L}{\partial \boldsymbol{h}(t)},$$

representing how the loss depends on the hidden state at time t. The time derivative is defined as follows:

$$\frac{d(\boldsymbol{a})}{dt} = -\boldsymbol{a}(t)^T \frac{\partial \boldsymbol{f}(t, \boldsymbol{h}(t), \boldsymbol{\theta}_t)}{\partial \boldsymbol{h}(t)}. \tag{13}$$

Given that Equation 13 is an ODE, the solution can be written as follows:

$$\frac{\partial L}{\partial \boldsymbol{h}(t)} = \int \boldsymbol{a}(t)^T \frac{\partial \boldsymbol{f}(t, \boldsymbol{h}(t), \boldsymbol{\theta}_t)}{\partial \boldsymbol{h}(t)} dt.$$

In order to calculate the gradient at time t_0, the integral is evaluated starting at initial point t_1:

$$\frac{\partial L}{\partial \boldsymbol{h}(t)} = \int_{t_1}^{t_0} \boldsymbol{a}(t)^T \frac{\partial \boldsymbol{f}(t, \boldsymbol{h}(t), \boldsymbol{\theta}_t)}{\partial \boldsymbol{h}(t)} dt.$$

In order to solve the gradients with respect to $\boldsymbol{\theta}$, the following ODE is solved:

$$\frac{\partial L}{\partial \boldsymbol{\theta}} = \int_{t_1}^{t_0} \boldsymbol{a}(t)^T \frac{\partial \boldsymbol{f}(t, \boldsymbol{h}(t), \boldsymbol{\theta}_t)}{\partial \boldsymbol{h}(t)} dt.$$

All in all, the adjoint method is used to backpropogate through the ODE by optimizing choices of the free parameters t_0, t_1, and $\boldsymbol{\theta}$ [38].

4 Experiments

The experiments focused on data-driven discovery of neural ODEs. First, observed data O was generated by solving an ODE with fixed initial conditions and parameters. The data was sampled at discrete, uniform time steps over a predetermined range, represented as

$$T = \{t_1, t_2, \ldots, t_n\}.$$

where t_i represents the ith datapoint. The corresponding observed data at each time point t_i is defined as

$$O = \{\boldsymbol{O}_1, \boldsymbol{O}_2, \ldots, \boldsymbol{O}_n\}, \text{where } \boldsymbol{O}_i = \hat{\boldsymbol{y}}(t)_i,$$

and n equals the length of T.

Next, a neural network, $\boldsymbol{f}_\theta$ was defined to learn the ODE. This neural network was subsequently used to define a neural ODE, defined as follows:

$$\frac{d\boldsymbol{y}(t)}{dt} = \boldsymbol{f}_\theta(t, \boldsymbol{y}(t)),$$

where $\boldsymbol{y}(t)$ is the state of the system, and $\boldsymbol{\theta}$ are the learnable parameters of the neural network.

Next, we defined the loss function using the MSE, as follows:

$$\ell(\theta) = \frac{1}{n}\sum_{i=1}^{n}(\boldsymbol{F}(\boldsymbol{f}_\theta, T)_i - \boldsymbol{O}_i)^2, \tag{14}$$

where $\boldsymbol{F}(\boldsymbol{f}_\theta, s)_i$ is value at the i-th time point predicted by using an ODE solver to solve the neural ODE, and $\boldsymbol{O}_i$ is the actual observed data at the i-th time point.

Finally, we used different optimization algorithms, including *Gradient Descent*, *ADAM*, *BFGS*, *L-BFGS*, and *Progressive Batching L-BFGS*, to minimize the loss (14) in three different experiments, comparing algorithms on the metric of gradient calls versus iterations.

4.1 Experiment 1: Lotka-Volterra

The Lotka-Volterra model is a pair of first order, non-linear ordinary differential equations. The model describes the predation of one species by another. If $x(t)$ is the prey population, and $y(t)$ is the predator population at time t, then then model can be represented by the following equations:

$$\frac{dx}{dt} = \alpha x - \beta xy, \tag{15}$$

$$\frac{dy}{dt} = -\delta y + \gamma xy. \tag{16}$$

where α is the growth rate of the prey in the absence of predators, β is a proportionality constant that links the mortality of the prey with the number of prey and predators, δ is a proportionality constant that links the increase in predators to the number of prey and predators, and γ is a constant of mortality for predators [39].

The model produces predator-prey cycles, with growth in the predator populations trailing that in the prey populations. Figure 3 is a graph of an example solution, which exhibits this behavior.

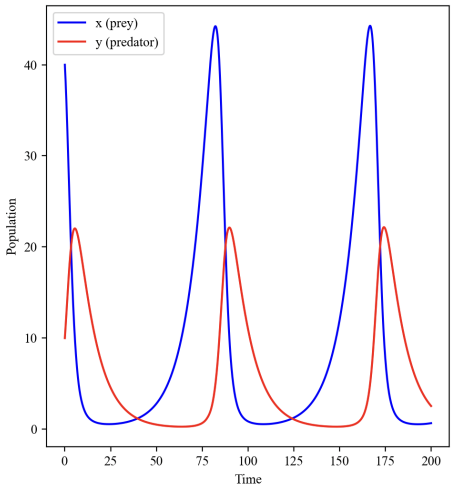

Figure 3: Periodic solutions for the prey $x(t)$ and the predator $y(t)$ for the Lotka-Volterra System, ((15), (16)) with initial conditions $x(0) = 40$ and $y(0) = 10$, and the parameters $\alpha = 0.1$, $\beta = 0.02$, $\gamma = 0.01$, and $\delta = 0.1$. Based on Figure 3.2 in [40].

4.1.1 Experimental Setup

The first Lotka-Volterra experiment followed a neural ODE architecture. The predator and prey equations were defined as follows:

$$\frac{du_1}{dt} = f_\theta^{(1)}(t, \boldsymbol{u}(t)), \tag{17}$$

$$\frac{du_2}{dt} = f_\theta^{(2)}(t, \boldsymbol{u}(t)). \tag{18}$$

The neural network $\boldsymbol{f}_\theta$ consists of two hidden layers. The first layer is a dense layer with 32 units and a hyperbolic tangent (tanh) activation function, and the second layer is a dense layer with 2 units and no activation function.

Figure 4 depicts the neural network architecture in a table format.

Layer Number	Layer Type	Parameters
1	Dense($2 \to 32$, tanh)	96
2	Dense($32 \to 2$)	66
Total Parameters		**162**

Figure 4: Network architecture illustrating a simple design with two layers and a total of 162 parameters.

Furthermore, the initial sample size, $|S_0|$, of the progressive batching L-BFGS method was set to 5, and the parameter θ of the IPQN test in Equation 10 was tuned to the value of 1.0.

4.1.2 Results

In Figure 5, we compare 5 methods on this problem, comparing the loss versus the number of gradient evaluations. As can be seen clearly from the graph, progressive baching L-BFGS, for a comparable number of iterations around 180 gradient evaluations, achieves a loss roughly three orders of magnitude smaller than the closest other method.

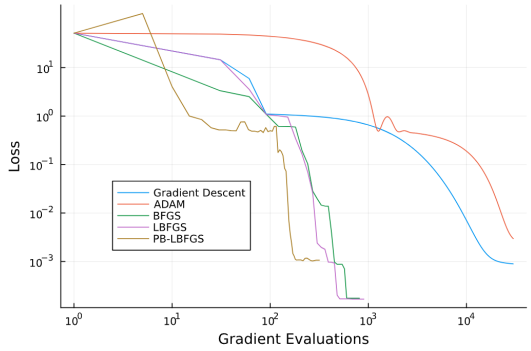

Figure 5: Comparison of optimization methods on Lotka-Volterra system with initial conditions $x(0) = 1$ and $y(0) = 1$, and parameters $\alpha = 0.75$, $\beta = 1.0$, $\gamma = 1.0$, and $\delta = 1.5$. Trained on 60 uniformly spaced time steps from [0,12].

The effects of loss are most clearly seen when plotting the learned trajectory. A lower loss indicates the dynamics are better approximated. In Figure 6, two graphs one presented. The first graph illustrates how the parameters learned by gradient descent after 180

iterations model the system, while the second graph shows the parameters learned by progressive batching L-BFGS over the same number of iterations. The gradient descent model performs poorly, with predicted points deviating significantly from the true data points. In contrast, the progressive batching L-BFGS model achieves a highly accurate representation of the system, with predicted points closely aligning with the true data points.

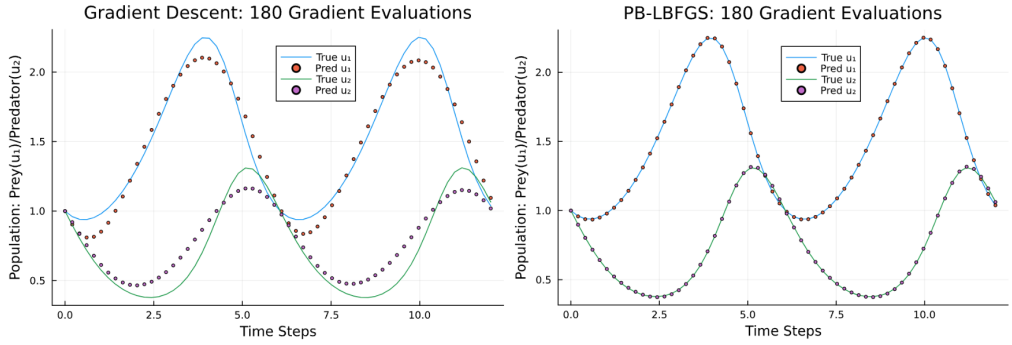

Figure 6: Comparison of learned parameter performance at 180 gradient evaluations for gradient descent and PB-LBFGS

4.2 Experiment 2: UDE approximation of Lotka-Volterra

The second experiment builds upon the first experiment's exploration of the Lotka-Volterra model by using a neural universal differential equation (UDE) architecture. The predator and prey equations were defined as follows:

$$\frac{du_1}{dt} = \alpha u_1 + f_\theta^{(1)}(t, \boldsymbol{u}(t)), \tag{19}$$

$$\frac{du_2}{dt} = \delta u_2 + f_\theta^{(2)}(t, \boldsymbol{u}(t)), \tag{20}$$

where $f_\theta^{(i)}(t, \boldsymbol{u}(t))$ is the ith component of $\boldsymbol{f}_\theta$. In this model, the α and δ parameters are known to the model, the neural network is used to learn the non-linear terms in equations (15) and (16). The neural network has three layers. The first layer of the neural network, $\boldsymbol{f}_\theta$, is defined by the following mapping:

$$L_1 : \begin{bmatrix} m_1 \\ m_2 \end{bmatrix} \mapsto \begin{bmatrix} m_1 \\ m_2 \\ m_1 m_2 \\ m_1^2 \\ m_2^2 \end{bmatrix}$$

The second and third layers are hidden layers. The second layer is a dense layer with 36 units and uses a tanh activation function. The third layer is a dense layer with 3 units and does not use an activation function. Figure 7 depicts the neural network architecture in a table format.

Layer Number	**Layer Type**	**Parameters**
1	$L_1(2 \to 5)$	5
2	Dense($5 \to 32$, tanh)	192
3	Dense($32 \to 2$)	66
Total Parameters		**263**

Figure 7: Neural network architecture detailing the layer configuration and associated parameter counts.

Additionally, the initial sample size, $|S_0|$, of the progressive batching L-BFGS method was set to 5, and the parameter θ of the IPQN test in Equation 10 was tuned to the value of 2.5.

4.2.1 Results

In Figure 8 we perform the same comparison as in Figure 5. Once again, progressive batching L-BFGS works well. Eventually at some point, the full-batch BFGS and L-BFGS methods find parameter values with a smaller loss out of all the methods. But progressive batching L-BFGS is competitive and significantly outperforms gradient descent and ADAM.

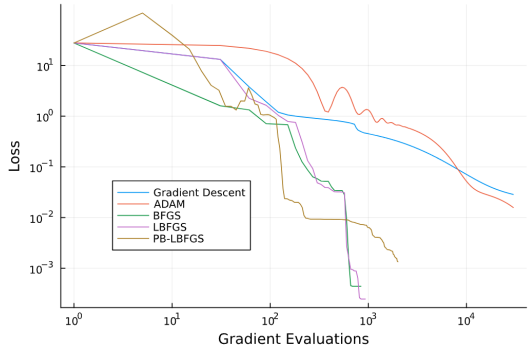

Figure 8: Comparison of optimization methods on Lotka-Volterra system with initial conditions $x(0) = 1$ and $y(0) = 1$, and parameters $\alpha = 0.75$, $\beta = 1.0$, $\gamma = 1.0$, and $\delta = 1.5$. Trained on 30 uniformly spaced time steps from [0,3].

4.3 Experiment 3: Non-linear Quantum Master Equation

The Bloch equation is a non-linear ODE that describes an open quantum system. In contrast to the Schrödinger's equation which describes a closed quantum system that is not affected by the environment. The Bloch accounts for the influence of the environment, specifically through temperature, which causes dissipation. This two-level system has successfully been applied to model spontaneous emission in quantum optics and nuclear magnetic resonance [41]. The Bloch Equation gives an accurate by phenomenological description of the time dependence of the magnetization vector $\vec{m}$. It is a non-linear coupled ODE, which can be represented by the following equation:

$$\frac{d\boldsymbol{m}}{dt} = \omega \begin{pmatrix} -m_2 \\ m_1 \\ 0 \end{pmatrix} - \gamma_0 \frac{k_B T_C}{\hbar \omega} - \gamma_0 \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} + \gamma_0 \frac{\mu(\boldsymbol{m})}{2} \begin{pmatrix} m_1 m_2 \\ m_2 m_3 \\ m_1^2 + m_2^2 + 2m_3^2 \end{pmatrix}, \tag{21}$$

where ω is the angular frequency associated with the energy difference between the two levels of the system, γ_0 is the spontaneous emmission rate, k_B is Boltzmann's constant, T_c is the temperature in Kelvin, and $\hbar$ is reduced Planck's constant [42].

4.3.1 Experiment

The Bloch equation experiment followed a neural universal differential equation (UDE) setup, defined as follows:

$$\frac{dm_1}{dt} = -\omega \cdot m_2 - \gamma_0 \cdot \alpha \cdot m_1 + f_\theta^{(1)}(t, \boldsymbol{y}(t)), \tag{22}$$

$$\frac{dm_2}{dt} = \omega \cdot m_1 - \gamma_0 \cdot \alpha \cdot m_2 + f_\theta^{(2)}(t, \boldsymbol{y}(t)), \tag{23}$$

$$\frac{dm_3}{dt} = -2\gamma_0 \cdot \alpha \cdot m_3 + f_\theta^{(3)}(t, \boldsymbol{y}(t)), \tag{24}$$

where $f_\theta^{(i)}(t, \boldsymbol{y}(t))$ is the ith component of $\boldsymbol{f}_\theta$. In this model, the neural network is used to learn the non-linear terms in equation (21). The neural network has three hidden layers.

The first layer of the neural network, $\boldsymbol{f}_\theta$, is defined by the following mapping:

$$L_1 : \begin{bmatrix} m_1 \\ m_2 \\ m_3 \end{bmatrix} \mapsto \begin{bmatrix} m_1 \\ m_2 \\ m_3 \\ m_1^2 \\ m_1 m_2 \\ m_1 m_3 \\ m_2^2 \\ m_2 m_3 \\ m_3^2 \end{bmatrix}$$

The second and third layers are hidden layers. The second layer is a dense layer with 36 units and uses a tanh activation function. The third layer is a dense layer with 3 units and does not use an activation function. Figure 9 depicts the neural network architecture in a table format.

Layer Number	**Layer Type**	**Parameters**
1	$L_1(3 \to 9)$	9
2	Dense($9 \to 36$, tanh)	360
3	Dense($36 \to 3$)	111
Total Parameters		**480**

Figure 9: Neural network architecture detailing the layer configuration and associated parameter counts.

Additionally, the initial sample size, $|S_0|$, of the progressive batching L-BFGS method was set to 5, and the parameter θ of the IPQN test in Equation 10 was tuned to the value of 2.5.

4.3.2 Results

In Figure 10 we perform the same comparison as in Figure 8 and 5. Progressive batching L-BFGS stagnates more than before, but nonetheless performs relatively well. L-BFGS is clearly the superior method for this problem, learning parameters that achieve a loss that is two orders of magnitude better than progressive batching L-BFGS. However, it is important to note that in the middle range of about 300 gradient evaluations, progressive batching L-BFGS achieves a lower loss compared to L-BFGS and the other three methods.

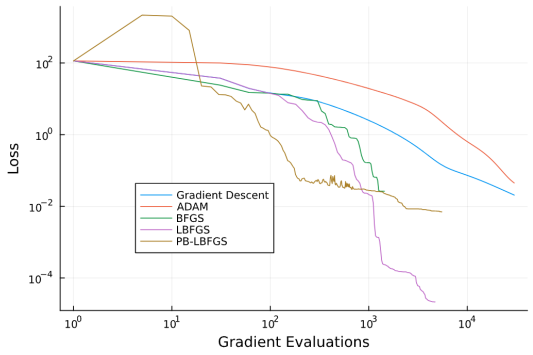

Figure 10: Comparison of optimization methods on Bloch equations with initial conditions $M_x = \frac{1}{\sqrt{3}}$, $M_y = \frac{1}{\sqrt{3}}$, and $M_z = \frac{1}{\sqrt{3}}$, parameters $\omega = 1.5$, $\gamma_0 = 0.1$, $K_B = 1.2$, $\hbar\omega = 1.01$ and $T_C = 0.02$. Trained on 30 uniformly spaced time steps from [0,3].

In Figure 11 the loss decreases rapidly with the initial batch size of 5. As the loss begins to plateau, the batch size increases to 6, but finds little luck learning better parameters; also, the loss oscillates significantly at this batch size, trying to pursue search directions that would yield better luck. Ultimately, it is not until the batch size is increased again to 17, when the loss decreases. From here, the batch size increases incrementally, contributing to a decrease in loss. When the batch size reaches the full batch count, the loss plateaus till convergence.

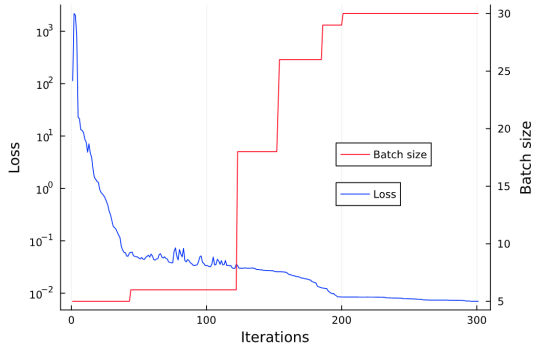

Figure 11: Progressive batching L-BFGS methods applied to Bloch equations, initialized 8 different IPQN test theta values. The graph displays the mean performance (solid line) across the runs, with the standard deviation represented as a shaded ribbon. The experimental conditions (initial conditions, parameters, etc.) are maintained from the experiment in Figure 10.

Figure 12 illustrates the average loss trajectory and standard deviation in 10 trials, each initialized with different randomly generated parameters. Tremendous variance occurs during the first 5 iterations, as the different trajectories experiment with search directions of various effectiveness. However, the variance significantly decreases as the loss slowly decreases, and the trajectories make steady progress to convergence.

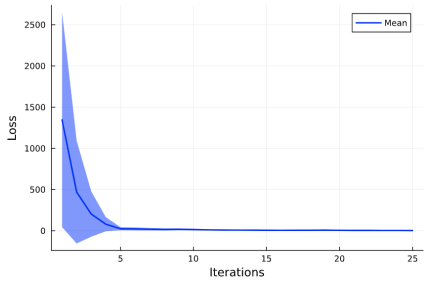

Figure 12: Progressive batching L-BFGS methods applied to Bloch equations, initialized with 5 different random starting parameter guesses. The graph displays the mean performance (solid line) across the runs, with the standard deviation represented as a shaded ribbon. The experimental conditions (initial conditions, parameters, etc.) are maintained from the experiment in Figure 10.

Figure 13 examines progressive batching L-BFGS and its performance on different values of $\boldsymbol{\theta}$, which is used to determine when the batch size changes in the IPQN test. Correlation between $\boldsymbol{\theta}$ and model performance can not be drawn. Therefore, tuning $\boldsymbol{\theta}$ is an important consideration that should be conducted on a problem by problem basis.

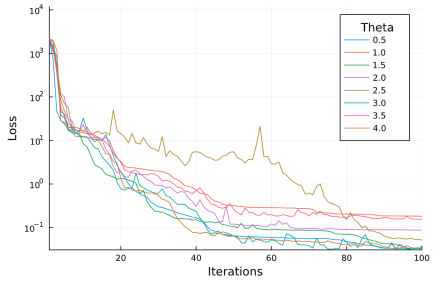

Figure 13: Progressive batching L-BFGS methods batch-size versus loss graphed over iterations. The experimental conditions (initial conditions, parameters, etc.) are maintained from the experiment in Figure 10.

5 Conclusion

In this thesis, progressive batching L-BFGS (PB-LBFGS) was compared to gradient descent, ADAM, BFGS, and L-BFGS in three experiments involving data-driven discovery with neural ODEs. The first experiment used neural ODEs to learn the Lotka-Volterra (LV) predator-prey ODEs. The second experiment extended the first experiment by applying neural UDEs to the same LV model. Finally, the third experiment used neural ODEs to learn the Bloch equation, a non-linear quantum master equation.

The results for experiments 1, 2 and 3 are given in Figures 5, 8 and 10, respectively, where the loss was plotted versus the gradient evaluations. In all experiments, the initial batch size of the PB-LBFGS method was initialized at $S_0 = 5$. The parameter θ of the IPQN test in Equation 10, which controls the increase in batch size in the PB-LBFGS method, was individually tuned for each experiment by choosing among the three values: 1, 2.5, 4.

The results showed that L-BFGS achieved the best loss, followed by BFGS, PB-LBFGS, GD, and ADAM. The PB-LBFGS method achieved a similar loss to GD. The PB-LBFGS method, however, achieved the fastest initial significant reduction in loss with respect to gradient evaluations. This is illustrated by Figure 6 which compares the learned trajectories of a model trained by gradient descent and PB-LBFGS at 180 gradient evaluations. The PB-LBFGS trained model demonstrates a superior approximation of the dynamics, with the learned trajectory closely modeling the true trajectory. However, the gradient descent trained model demonstrates a poor approximation of the dynamics, where the learned trajectory does not match the true trajectory.

Furthermore, examining Figure 11, it is evident that the PB-LBFGS method exhibits an efficient initial behavior. This is because the batch size is small at the beginning, enabling quick gradient calculations and frequent updates. When progress in decreasing loss stagnates, the algorithm automatically increases the batch size, helping the loss decrease further.

The numerical results presented in this paper, make a strong value proposition for the PB-LBFGS method. PB-LBFGS has fast convergence, superior generalization, and enables the exploitation of parallelism. Investigating the potential of PB-LBFGS as a viable alternative to first-order methods for deep learning applications merits further research.

References

[1] R. T. Chen, Y. Rubanova, J. Bettencourt, and D. K. Duvenaud, "Neural ordinary differential equations," *Advances in neural information processing systems*, vol. 31, 2018.

[2] S. J. Wright, "Numerical optimization," 2006.

[3] J. E. Dennis, Jr and J. J. Moré, "Quasi-newton methods, motivation and theory," *SIAM review*, vol. 19, no. 1, pp. 46–89, 1977.

[4] H. Robbins and S. Monro, "A stochastic approximation method," *The Annals of Mathematical Statistics*, vol. 22, no. 3, pp. 400–407, 1951.

[5] R. Bollapragada, J. Nocedal, D. Mudigere, H.-J. Shi, and P. T. P. Tang, "A progressive batching l-bfgs method for machine learning," in *International Conference on Machine Learning*, pp. 620–629, PMLR, 2018.

[6] R. H. Byrd, G. M. Chin, J. Nocedal, and Y. Wu, "Sample size selection in optimization methods for machine learning," *Mathematical programming*, vol. 134, no. 1, pp. 127–155, 2012.

[7] M. P. Friedlander and M. Schmidt, "Hybrid deterministic-stochastic methods for data fitting," *SIAM Journal on Scientific Computing*, vol. 34, no. 3, pp. A1380–A1405, 2012.

[8] R. Bollapragada, R. Byrd, and J. Nocedal, "Adaptive sampling strategies for stochastic optimization," *SIAM Journal on Optimization*, vol. 28, no. 4, pp. 3312–3343, 2018.

[9] S. Smith, "Don't decay the learning rate, increase the batch size," *arXiv preprint arXiv:1711.00489*, 2017.

[10] A. Devarakonda, M. Naumov, and M. Garland, "Adabatch: Adaptive batch sizes for training deep neural networks," *arXiv preprint arXiv:1712.02029*, 2017.

[11] S. De, A. Yadav, D. Jacobs, and T. Goldstein, "Automated inference with adaptive batches," in *Artificial Intelligence and Statistics*, pp. 1504–1513, PMLR, 2017.

[12] L. Balles, J. Romero, and P. Hennig, "Coupling adaptive batch sizes with learning rates," *arXiv preprint arXiv:1612.05086*, 2016.

[13] J. P. Crutchfield and B. McNamara, "Equations of motion from a data series ‘," *Complex systems*, vol. 1, pp. 417–452, 1987.

[14] M. Schmidt and H. Lipson, "Distilling free-form natural laws from experimental data," *science*, vol. 324, no. 5923, pp. 81–85, 2009.

[15] J. Bongard and H. Lipson, "Automated reverse engineering of nonlinear dynamical systems," *Proceedings of the National Academy of Sciences*, vol. 104, no. 24, pp. 9943–9948, 2007.

[16] R. González-García, R. Rico-Martìnez, and I. G. Kevrekidis, "Identification of distributed parameter systems: A neural net based approach," *Computers & chemical engineering*, vol. 22, pp. S965–S968, 1998.

[17] A. Cauchy *et al.*, "Méthode générale pour la résolution des systemes d'équations simultanées," *Comp. Rend. Sci. Paris*, vol. 25, no. 1847, pp. 536–538, 1847.

[18] J. E. Dennis Jr and R. B. Schnabel, *Numerical methods for unconstrained optimization and nonlinear equations*. SIAM, 1996.

[19] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization 3rd international conference on learning representations," in *ICLR 2015-Conference Track Proceedings*, vol. 1, 2015.

[20] B. T. Polyak, "Some methods of speeding up the convergence of iteration methods," *Ussr computational mathematics and mathematical physics*, vol. 4, no. 5, pp. 1–17, 1964.

[21] G. Goh, "Why momentum really works," *Distill*, vol. 2, no. 4, p. e6, 2017.

[22] S. Ruder, "An overview of gradient descent optimization algorithms," *arXiv preprint arXiv:1609.04747*, 2016.

[23] R. Elshamy, O. Abu-Elnasr, M. Elhoseny, and S. Elmougy, "Improving the efficiency of rmsprop optimizer by utilizing nestrove in deep learning," *Scientific Reports*, vol. 13, no. 1, p. 8814, 2023.

[24] G. Hinton, "Neural networks for machine learning," 2012.

[25] M. Kochenderfer, *Algorithms for Optimization*. The MIT Press Cambridge, 2019.

[26] C. G. Broyden, "Quasi-newton methods and their application to function minimisation," *Mathematics of Computation*, vol. 21, no. 99, pp. 368–381, 1967.

[27] P. Wolfe, "Convergence conditions for ascent methods," *SIAM review*, vol. 11, no. 2, pp. 226–235, 1969.

[28] C. G. Broyden, "The convergence of a class of double-rank minimization algorithms 1. general considerations," *IMA Journal of Applied Mathematics*, vol. 6, no. 1, pp. 76–90, 1970.

[29] R. Fletcher, "A new approach to variable metric algorithms," *The computer journal*, vol. 13, no. 3, pp. 317–322, 1970.

[30] D. Goldfarb, "A family of variable-metric methods derived by variational means," *Mathematics of computation*, vol. 24, no. 109, pp. 23–26, 1970.

[31] D. F. Shanno, "Conditioning of quasi-newton methods for function minimization," *Mathematics of computation*, vol. 24, no. 111, pp. 647–656, 1970.

[32] D. C. Liu and J. Nocedal, "On the limited memory bfgs method for large scale optimization," *Mathematical programming*, vol. 45, no. 1, pp. 503–528, 1989.

[33] G. Birkhoff and G.-C. Rota, *Ordinary Differential Equations*. New York: John Wiley & Sons, 4 ed., 1989.

[34] J. Orloff, "Supplementary notes for 18.03: Differential equations." `https://math.mit.edu/~jorloff/suppnotes/suppnotes03/1803SupplementaryNotes_full.pdf`, 2017.

[35] M. A. Nielsen, *Neural Networks and Deep Learning*. Determination Press, 2015. Online; accessed 21-December-2024.

[36] CS231n Convolutional Neural Networks for Visual Recognition, "Neural networks part 1: Setting up the architecture." `https://cs231n.github.io`, n.d. Accessed: December 19, 2024.

[37] GeeksforGeeks, "Backpropagation in neural network." `https://www.geeksforgeeks.org/backpropagation-in-neural-network/`, 2024. Last Updated: 02 Nov, 2024.

[38] J. Sinai, "Understanding neural odes." `https://jontysinai.github.io/jekyll/update/2019/01/18/understanding-neural-odes.html`, 2019. Accessed: December 19, 2024.

[39] P. J. Wangersky, "Lotka-volterra population models," *Annual Review of Ecology and Systematics*, vol. 9, pp. 189–218, 1978.

[40] J. Murray, "Models for interacting populations," *Mathematical biology: I. An introduction*, pp. 79–118, 2002.

[41] E. L. Hahn, "Concepts of nmr in quantum optics," *Concepts in Magnetic Resonance: An Educational Journal*, vol. 9, no. 2, pp. 69–81, 1997.

[42] H. C. Öttinger, "Nonlinear thermodynamic quantum master equation: Properties and examples," *Physical Review A—Atomic, Molecular, and Optical Physics*, vol. 82, no. 5, p. 052119, 2010.