

**OPTIMIZING NEURAL ORDINARY DIFFERENTIAL
EQUATIONS WITH LOOKAHEAD OPTIMIZER**

**NIYATA SANNGAI
STUDENT ID: 630510482**

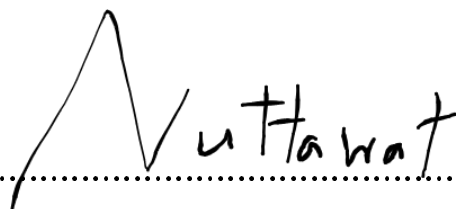
**DEPARTMENT OF MATHEMATICS, FACULTY OF SCIENCE
CHIANGMAI UNIVERSITY
SEMESTER 1, ACADEMIC YEAR 2023**

OPTIMIZING NEURAL ORDINARY DIFFERENTIAL EQUATIONS
WITH LOOKAHEAD OPTIMIZER

NIYATA SANNGAI

A PROJECT HAS BEEN APPROVED
TO BE A PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF BACHELOR OF SCIENCE
IN MATHEMATICS

EXAMINING COMMITTEE:

.......... CHAIRPERSON

(Asst. Prof. Dr. Nuttawat Sontichai)

.......... MEMBER

(Asst. Prof. Dr. Hatairat Yingtaweessittikul)

October 4, 2023

OPTIMIZING NEURAL ORDINARY DIFFERENTIAL EQUATIONS
WITH LOOKAHEAD OPTIMIZER

NIYATA SANNGAI
STUDENT ID: 630510482

A PROJECT HAS BEEN APPROVED
TO BE A PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF BACHERLOR OF SCIENCE
IN MATHEMATICS
CHIANG MAI UNIVERSITY

2023

ACKNOWLEDGEMENTS

I would like to express my sincere thanks to my thesis advisor, Dr. Nuttawat Sontichai for his invaluable help and constant encouragement throughout this research. I very much appreciate him for his support and advice. I would not have achieved this far and this project would not have been completed without all the care and support that I have always received from him. And also Dr. Ovidui Bagdasar for giving me an opportunity to working with him and for all his advice and support while I was in the United Kingdom. The time for living in the United Kingdom passed smoothly because of his kind.

Next, I sincerely thank Development and Promotion of Science and Technology Talents Project scholarship for giving me a great opportunity. this Research may not take place, if I receive financial assistance from them.

I would also like to thank all my friends both in Thailand and the UK for having my back and for their encouragement. You all are really making my time memorable and I appreciate that a lot.

Finally, I would like to thank you to myself for not giving up. Even though I'm tired and discouraged, I still keep doing this project until I make it. So I hope that this research will be useful in the future.

NIYATA SANNGAI

Title OPTIMIZING NEURAL ORDINARY DIFFERENTIAL EQUATIONS
WITH LOOKAHEAD OPTIMIZER

Author NIYATA SANNGAI

Student ID: 630510482

Advisory Committee Asst. Prof. Dr. Nuttawat Sontichai

Member Asst. Prof. Dr. Hatairat Yingtaweesittikul

Abstract

This study delves into the application of Neural Ordinary Differential Equations (Neural ODEs) within the machine learning domain, introducing an innovative optimization strategy using the Lookahead Optimizer. Utilizing Euler's Method for both forward and adjoint sensitivity calculations, we discuss the trade-offs between computational efficiency and numerical accuracy. Our experimental results indicate that the Lookahead Optimizer converges faster at lower learning rates and exhibits greater stability at higher learning rates compared to standard Gradient Descent. This work provides empirical evidence supporting the efficacy of the Lookahead Optimizer in Neural ODE contexts and offers an open-source codebase for future research.

Contents

ACKNOWLEDGEMENT	i
ABSTRACT	ii
CONTENTS	iii
1 Introduction	1
2 Mathematical Background	3
2.1 Chain Rules	3
2.2 Neural Network	3
2.3 Deep Learning Neural Network	6
2.4 Xavier method	7
2.5 Gradient Descent algorithm	8
2.6 Residual Neural Networks (ResNets)	9
2.7 Euler's Method	11
2.8 Big O notation	12
3 Methodology	13
3.1 Neural Ordinary Differential Equations (NODEs)	13
3.2 Lookahead Optimizer: k steps forward, 1 step back	19
3.3 Improving Adjustment of NODEs's Weights	20
4 Experimental Study	23

5 Conclusion	33
References	35
Appendix	37

Chapter 1

Introduction

Ordinary Differential Equations (ODEs) have long been fundamental in applied mathematics, with applications in diverse fields such as physics, engineering, biology, and economics [1]. The recent introduction of Neural Ordinary Differential Equations (Neural ODEs) has extended the reach of ODEs into machine learning and data science, especially in areas like time-series modeling and generative modeling [2, 3]. However, the optimizing of Neural ODEs presents unique challenges, including issues like vanishing gradients and stiff ODEs, which often make traditional optimization algorithms like Stochastic Gradient Descent (SGD) and Adam ineffective [4, 5].

In this study, we introduce a pioneering method for training Neural ODEs with fixed initial and target states. Our approach encompasses implementing a Neural ODE with a sigmoid activation function, utilizing the Euler forward method [12] for its solution, and exploring the impact of various optimization techniques, notably standard gradient descent and the Lookahead optimizer. Additionally, we adopt the Glorot (Xavier) weight initialization [5] to ensure stable initial weights and maintain fixed parameters such as time intervals, maximum time, and stopping criteria. This comprehensive strategy is further bolstered by our experiments with distinct hyperparameters, including learning rate, time points, and iterations for both standard gradient descent and lookahead.

The primary contributions of this study are twofold: we provide empirical evidence for the efficacy of the Lookahead Optimizer in the context of Neural ODEs contexts and we offer an accessible and implement codebase, which will be publicly available for further academic exploration.

This study primarily aims to investigate the performance of the Lookahead Optimizer within Neural ODEs contexts and to offer an open-source, user-friendly codebase. This will serve as a practical guide for those new to Neural ODEs and keen on further exploration.

Expected benefits include an enhanced understanding of Neural ODE optimization and

educational value, especially for students and educators in applied mathematics and machine learning.

Scope of the Study

1. **Optimization Algorithms:** The study will compare the Lookahead Optimizer with traditional method, excluding other optimization algorithms.
2. **Numerical Methods:** The focus will be on Euler's Method for ODE solutions. Other techniques like Runge-Kutta will not be explored.
3. **Experimental Validation:** Validation will be base on predefined experiments to test the Lookahead Optimizer's efficacy in Neural ODE contexts. These may not encompass all possible Neural ODE applications or configurations.

This report is organized into sections covering Mathematical Background, Methodology, Experimental Study, Conclusion and Discussion. The latter will expand on avenues for future research, detailing potential research directions and their significance.

Chapter 2

Mathematical Background

2.1 Chain Rules

For any real-valued function f composed of the other real functions $p_i(x)$, when $i = 1, 2, \dots, n$ such that $f = p_n(x) \circ p_{n-1}(x) \circ \dots \circ p_2(x) \circ p_1(x)$, the rate of change of f with respect to x is expressed as [7]

$$\frac{df}{dx} = \frac{df}{dp_n} \frac{dp_n}{dp_{n-1}} \dots \frac{dp_2}{dp_1} \frac{dp_1}{dx}$$

2.2 Neural Network

Neural Network (NN) is interconnected layers of small units called nodes that perform mathematical operations to detect patterns in data. NN algorithms are built in a way that mimics how human neurons work.

Firstly, we will introduce key terms used when discussing Neural Networks.

1. Neuron or node: a basic building block of a NN. It takes weighted values, performs mathematical calculations and produces output. It is also called a unit, node or perceptron.

2. Input layer: The layer of information or data from the outside world enters the neural network.

3. Hidden layers: The layer that take input from the input layer or other hidden layers. Each hidden layer analyzes the output from the previous layer, processes it further, and passes it on to the next layer.[8]

4. Output Layer: The layer which give the final result of all the data processing by the network.

5. Weights: These values explain the strength (degree of importance) of the connection

between any two nodes.

6. Bias: is a constant value added to the sum of the weighed sum of the inpts. It is used to accelerate or delay the activation of a given node.

7. Activation function: is a function used to introduce the non-linearity phenomenon into the neural network system. This property enables the network to learn more complex patterns, for example

a sigmoid function, a function with a characteristic S-shaped curve.

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

Domain of a sigmoid function is between $(-\infty, \infty)$ and range is between $(0, 1)$. Therefore, it is especially used in binary classification or in models where we have to predict the probability as an output.

a hyperbolic tangent function is also have S-shaped like logistic sigmoid function.

$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

Domain of a *tanh* function is also $(-\infty, \infty)$ but the range of the *tanh* function is $(-1, 1)$. So it gave better performance than a sigmoid function from having larger gradient.

a ReLU (Rectified Linear Unit) Activation Function

$$f(x) = \max(0, x)$$

Domain of a ReLU function is $(-\infty, \infty)$ and the range of the function is $(0, x)$. The gradient of ReLU is 1 for positive values input and 0 for negative values input.

8. Deep Neural Network (DNN): These are an neural networks with many hidden layers.

Next, we will explain how does it work by considering simplest neural network called perceptron, Figure 1. A perceptron was invented by Frank Rosenblatt at the Cornell Aeronautical Laboratory in 1957. A perceptron has one or more than one inputs, a process, and only one output. It is used as an algorithm or a linear classifier to facilitate supervised learning of binary classifiers. the perceptron relies on a linear predictor function combining weights and bias to make its predictions. Its predictions are based on a combination that includes weights and bias by algorithm below.

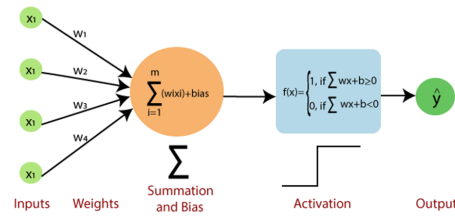


Figure 1 : Simply Neural Networks Architecture

<https://www.nomidl.com/wp-content/uploads/2022/04/image-5.png>

Perceptron algorithm

Feed-forward

Input State : Require input X , initial weight w_0 , initial bias b_0

Hidden State : $z(t) = w_0 X + b_0$

if $z(t) > 0$: $h(t) = 1$

else : $h(t) = 0$

Output State : Return $h(t)$

backpropagation

Require true values Y , learning rate η

Calculate Loss : $L = (h(t) - Y)^2$

Calculate Gradient : $\frac{\partial L}{\partial W_0}, \frac{\partial L}{\partial b_0}$

Update Weight and Bias :

$$w_0 = w_0 + \eta \frac{\partial L}{\partial W_0}$$

$$b_0 = b_0 + \eta \frac{\partial L}{\partial b_0}$$

then keep repeating the algorithm untill loss value reach the setisfied error.

2.3 Deep Learning Neural Network

Deep learning is a subset of machine learning and which is essentially a neural network with multiple layers. It distinguishes itself from classical machine learning through the type of data it processes and the methods it employs for learning. Deep learning neural networks attempt to mimic the human brain by using a combination of data inputs, weights, and biases. These elements work in concert to accurately recognize, classify, and describe data.

Deep neural networks comprise of multiple layers of interconnected nodes, Each layers builds upon the previous one to refine and optimize predictions or categorizations. This sequence of computations, which processes through the input, hidden, and output layers, is known as feed-forward propagation.

Feedforward (FNN) Algorithm

In neural networks with t hidden layers, we define the variables as follow

$$x_t = \begin{pmatrix} x_1 \\ \vdots \\ x_n \\ 1 \end{pmatrix}, \quad z_t = \begin{pmatrix} z_1 \\ \vdots \\ z_n \\ 1 \end{pmatrix}, \quad \text{and} \quad h_t = \begin{pmatrix} h_1 \\ \vdots \\ h_n \\ \sigma(1) \end{pmatrix}$$

when x_t , z_t , and h_t is the input vector, the value from hidden layers, and the vector at time t , where $t = 1, 2, \dots$ in order.

The following equations describe the feedforward process

$$z_{t+1} = w_t \cdot h_{t-1} \tag{2.1}$$

$$h_t = \sigma(z_t) \tag{2.2}$$

here $w_t = \begin{pmatrix} w_1 \\ \vdots \\ w_n \\ b \end{pmatrix}$ represent the weigh, and σ is the activate function.

Another essential process is backpropagation. This algorithm adjusts the weights and biases by calculating the derivative of the errors in predictions with respect to the weights it dose so by moving backward through the layers, as shown below.

Backpropagation Algorithm

We define the error as the loss function $L(h_t) = \sigma(z_t)$. the weight update equation is given by :

$$w_{t+1} = w_t - \eta \frac{\partial L}{\partial w_t}$$

when η is leraning rate for updating weight.

Together, feedforward propagation and backpropagation enable a neural network to make predictions and adjust for errors. As the algorithm iterates, its accuracy improves gradually.

2.4 Xavier method

For starters, an improper initialization would lead to an unoptimizable model. Choosing the right initializer is an essential step in training to maximize performance. For example, if you go with the Xavier initialization, you must ensure this technique is appropriate for your model. In addition, initializing weighted neural networks also shortens the convergence time and minimizes the loss function.

The Xavier initialization[\[5\]](#) is a popular technique for initializing weights in a neural network. It is a state-of-the-art technique with which anyone interested in neural networks

should be sufficiently acquainted. In the field of deep learning, we use the Xavier method to initialize the weights of neural networks to mitigate the problem of vanishing gradients and exploding gradients. This introduced method in 2010. The main purpose of initializing weights through the Xavier method is to propagate effectively during forward and backward propagation.

2.5 Gradient Descent algorithm

Gradient descent [14] is one of the most popular algorithm stoper for optimization because of its simplicity. Gradient descent is a way to minimize an objective function $J(\theta)$ parameterized by a models parameters $\theta \in \mathbb{R}^d$ by updating the parameters in the opposite direction of the gradient of the objective function $\nabla J(\theta)$ with respect to the parameters. The learning rate η determines the size of the steps we take to reach a local minimum. In other words, we follow the direction of the slope of the surface created by the objective function downhill until we reach a local minimum point.

There are three variants of gradient descent, which differ in how much data we use to compute the gradient of the objective function.

1. Batch gradient descent

batch gradient descent computes the gradient of the cost function with respect to the parameters θ for the entire training data set

$$\theta = \theta - \eta \nabla J(\theta)$$

As we need to calculate the gradients for the whole data set to perform one update. So batch gradient descent can be very slow and is intractable for data sets that do not fit in memory. Batch gradient descent is guaranteed to converge to the global minimum for convex error surfaces and to a local minimum for non-convex surfaces.

2. Stochastic gradient descent (SGD)

Stochastic gradient descent(SGD) in contrast performs a parameter update for each training example x_i and label y_i

$$\theta = \theta - \eta \nabla J(\theta, x_i, y_i)$$

SGD performing one update at a time. It is therefore usually much faster and can also be used to learn online. Due to SGD performs frequent updates, it has a high variance that cause the objective function to fluctuate heavily. While batch gradient descent converges to the local minimum of the parameters are placed in, SGD's fluctuation, on the one hand, enables it to jump to new and potentially better local minimum point. However, it has been shown that when we slowly decrease the learning rate, SGD shows the same convergence behaviour as batch gradient descent, almost certainly converging to a local or the global minimum for non-convex and convex optimization respectively.

3. Mini-batch gradient descent

Mini-batch Gradient Descent finally takes the best of both worlds and performs an update for every mini-batch of n training examples

$$\theta = \theta - \eta \nabla J(\theta, x_{i:i+1}, y_{i:i+1})$$

This way, it reduces the variance of the parameter updates, which can lead to more stable convergence and can make use of highly optimized matrix optimizations. gradient descent is typically the algorithm of choice when training a neural network and the term SGD usually is employed also when mini-batches are used.

2.6 Residual Neural Networks (ResNets)

Residual Networks or ResNets, are a specific type of neural network introduced in 2015 by Kaiming He, Xiangyu Zhang, Shaoqing Ren and Jian Sun in their paper Deep Residual Learning for Image Recognition . [\[13\]](#)

One of the most effective properties of ResNets is their ability to mitigate the vanishing gradient problem commonly encountered in deep learning architectures.

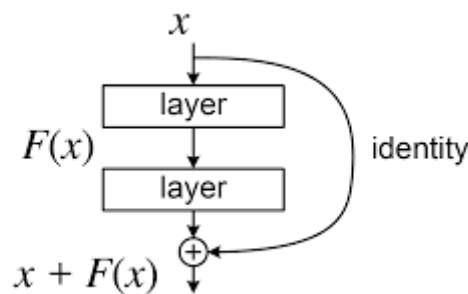


Figure 2 : ResNets Architecture

The ResNets architecture can be described by a sequence of transformations to the hidden state as follow :

$$h_t = \sigma(z_t) + z_t \quad (2.3)$$

$$z_t = w_t \cdot h_{t-1} \quad (2.4)$$

Before delving into how ResNets solve the vanishing gradient problem, It's important to note that the issue arises when the gradients of the loss function with respect to the weights of the early layers become vanishingly small. As a result, these layers receive little to no updates during backpropagation, leading to slow convergence or even stagnation.

In a standard Artificial Neural Network, the iteration can be described by (2.1) and (2.2)

Using the chain rule, we find

$$\begin{aligned} \frac{\partial L(h_t)}{\partial w_t} &= \frac{\partial L(h_t)}{\partial h_t} \frac{\partial h_t}{\partial w_t} \\ &= \frac{\partial L(h_t)}{\partial h_t} \frac{\partial h_t}{\partial z_t} \frac{\partial z_t}{\partial w_t} \\ &= \frac{\partial L(h_t)}{\partial h_t} \sigma'(z_t) h_{t-1} \end{aligned}$$

In deep networks, this can be further expanded as

$$\begin{aligned} \frac{\partial L(h_1)}{\partial w_1} &= \frac{\partial L(h_t)}{\partial h_t} \frac{\partial h_t}{\partial h_{t-1}} \frac{\partial h_{t-1}}{\partial h_{t-2}} \cdots \frac{\partial h_2}{\partial h_1} \frac{\partial h_1}{\partial z_1} \frac{\partial z_1}{\partial w_1} \\ &= \frac{\partial L(h_t)}{\partial h_t} \sigma'(z_t) w_t \sigma'(z_{t-1}) w_{t-1} \cdots \sigma'(z_3) w_3 \sigma'(z_2) w_2 \sigma'(z_1) h_0 \\ &= \frac{\partial L(h_t)}{\partial h_t} \sigma'(z_t) \sigma'(z_{t-1}) \cdots \sigma'(z_2) \sigma'(z_1) w_t w_{t-1} \cdots w_3 w_2(x) \\ &= \frac{\partial L(h_t)}{\partial h_t} \sigma'(z_t) \left(\prod_{i=1}^{t-1} \sigma'(z_i) w_{i+1} \right) x \end{aligned}$$

While using gradient base activate function, there are also cases where $\prod_{i=k}^{t-1} \sigma'(h_i)$ approaches zero, especially when $k \in \mathbb{N}$ and $k \leq t - 1$. In such cases, $\partial L(h_k)/\partial w_k$ also converges to zero. This result in minimal height updates in the early layers, contributing to

the vanishing gradient problem.

In ResNets, the sequence of transformations to the hidden state can also be described by (2.3) and (2.4). The gradient can be computed as

$$\begin{aligned}\frac{\partial L(h_1)}{\partial w_1} &= \frac{\partial L(h_t)}{\partial h_t} \frac{\partial h_t}{\partial h_{t-1}} \frac{\partial h_{t-1}}{\partial h_{t-2}} \dots \frac{\partial h_2}{\partial h_1} \frac{\partial h_1}{\partial z_1} \frac{\partial z_1}{\partial w_1} \\ &= \frac{\partial L(h_t)}{\partial h_t} [\sigma'(z_t) + 1] w_t [\sigma'(z_{t-1}) + 1] w_{t-1} \dots [\sigma'(z_1) + 1] w_1(x) \\ &= \frac{\partial L(h_t)}{\partial h_t} \left(\prod_{i=1}^t [\sigma'(z_i) + 1] w_i \right) x\end{aligned}$$

See that even though $\sigma'(z_i)$ lead to zero, term $\prod_{i=1}^t [\sigma'(z_i) + 1]$ still not converge to zero. So here was how ResNets solve a vanishing gradient problem.

2.7 Euler's Method

Euler's Method [12] is the most basic explicit method for the numerical integration of ordinary differential equations (ODEs) with a given initial value. When defining an ODE of order N as

$$y^{(N)}(x) = f(x, y(x), y'(x), \dots, y^{(N-1)}(x))$$

The process begins with an initial value \vec{x}_0 by setting step size h and

$$\vec{y}_0 = \vec{y}(\vec{x}_0)$$

The variable are then updated until a satisfied error level is reached, according to

$$\vec{x}_{i+1} = \vec{x}_i + h$$

and

$$\vec{y}_{i+1} = \begin{pmatrix} y_{i+1} \\ y'_{i+1} \\ \vdots \\ y_{i+1}^{(N-1)} \\ y_{i+1}^{(N)} \end{pmatrix} = \begin{pmatrix} y_i + hy'_i \\ y'_i + hy''_i \\ \vdots \\ y_i^{(N-1)} + hy^{(N)}_i \\ y_i^{(N)} + hf(x_i, y_i(x), y'_i(x), \dots, y_i^{(N-1)}(x)) \end{pmatrix}$$

The above formula is implemented iteratively until an approximation of the solution to the ODEs is reached.

2.8 Big O notation

Big O notation (\mathcal{O}) is a mathematical notation that describes the limiting behavior of a function when the argument tends towards a particular value or infinity. The letter O stand for Ordnung, meaning the order of approximation.

In computer science, big O notation is used to classify algorithms according to how their run time or space requirements grow as the input size grows.

Big O can also be used to describe the error term in an approximation to a mathematical function. The most significant terms are written explicitly, and then the least-significant terms are summarized in a single big O term. Consider, for example, the exponential series

$$\begin{aligned} e^x &= 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots \\ &= 1 + x + \frac{x^2}{2!} + \mathcal{O}(x^3) \\ &= 1 + x + \mathcal{O}(x^2) \end{aligned}$$

Chapter 3

Methodology

3.1 Neural Ordinary Differential Equations (NODEs)

Neural Ordinary Differential Equations (NODEs) represent a new family of deep neural network introduced by Ricky chen, Yulia Rubanova, Jesse Bettencourt, and David Duvenaud [2] this work recived the "Best Paper Awards" at the Neural Information Processing Systems (NeurIPS) conference in 2018 due to its innovative perspective on neural network architecture.

In NODEs, rather than specifying a discrete sequence of hidden layers, the model's parameterize the derivative of the hidden layer states to create continuous-depth models. This allows to training of models using ODEs, considering forward propagation in a neural network as the one-step discretization of an ODE. Starting with ResNets, the model composes a sequence of tranformations to the hidden state in (2.3), these the model similar to Euler's Method for first-order ODEs. Subsequently the model modulates ResNets by adding layers and taking a smaller steps, then parameterizes the continuous dynamics of hidden units using an ordinary differential equation in form

$$\frac{dh_t}{dt} = f(h_t, t, \theta)$$

For the feedforward algorithm, the model starts from input layer h_0, X , and defines the output layer h_T as the solution. This value can be computed by a black-box differential equation solver, which evaluates the hidden unit dynamics f wherever necessary to achieve the desired accuracy. This distinguishes ResNets and NODEs as shown in Figure 3.

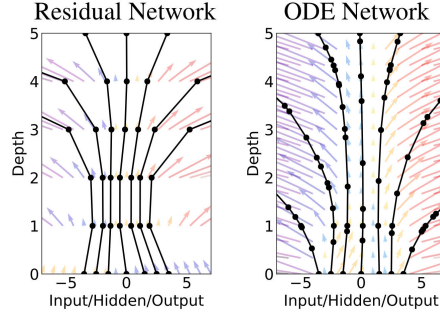


Figure 3 : Contrasts between two approaches.

Soucre : Neural Ordinary Differential Equations. [2]

the main difficulty in training continuous- depth network lies in performing back-propagation through the ODE solver. While, calculating the derivative through the operation of feedforward is strightforward , it incurs a high memory costs and introduces additional numerical errors.

Therefore, the ODE solver is treated as a black box. Gradients are cumputed using the adjoint sensitivity method, which solves a second, augmented ODE backward in time. This method is applicable to all ODE solvers, scales linearly with problem size, has low memory costs, and controls numerical errors effectively.

The loss function is defined as

$$L(z(t_1)) = L\left(z(t_0) + \int_{t_0}^{t_1} f(z(t), t, \theta dt)\right) = L(\text{ODESolve}(z(t_0), f, t_0, t_1, \theta))$$

gradients with respect to θ are required. The first step is to assign the gradient of the loss with respect to hidden state $z(t)$ at each time t . This quantity is defined as the adjoint $a(t) = \partial L / \partial z(t_0)$.

Consider

$$a(t) = \frac{dL}{dz(t)}$$

In standard neural networks, the gradient of hidden layer h_t depends on the gradient of hidden layer h_{t+1} by chain rule

$$\frac{dL}{dh_t} = \frac{dL}{dh_{t+1}} \frac{dh_{t+1}}{dh_t}$$

With a continuous hidden state, we can write the transformation after an ε change

in time as

$$z(t + \varepsilon) = \int_t^{t+\varepsilon} f(z(t), t, \theta) dt + z(t) = T_\varepsilon(z(t), t)$$

and chain rule can also be applied

$$\frac{dL}{dz(t)} = \frac{dL}{dz(t + \varepsilon)} \frac{dz(t + \varepsilon)}{dz(t)} \quad \text{or} \quad a(t) = a(t + \varepsilon) \frac{\partial T_\varepsilon(z(t), t)}{\partial z(t)} \quad (3.1)$$

then consider

$$\begin{aligned} \frac{da(t)}{dt} &= \lim_{\varepsilon \rightarrow 0^+} \frac{a(t + \varepsilon) - a(t)}{\varepsilon} \\ &= \lim_{\varepsilon \rightarrow 0^+} \frac{a(t + \varepsilon) - a(t + \varepsilon) \frac{\partial}{\partial z(t)} T_\varepsilon(z(t))}{\varepsilon} \\ &= \lim_{\varepsilon \rightarrow 0^+} \frac{a(t + \varepsilon) - a(t + \varepsilon) \frac{\partial}{\partial z(t)} (z(t) + \varepsilon f(z(t), t, \theta) + \mathcal{O}(\varepsilon^2))}{\varepsilon} \\ &= \lim_{\varepsilon \rightarrow 0^+} \frac{a(t + \varepsilon) - a(t + \varepsilon) \left(I + \varepsilon \frac{\partial f(z(t), t, \theta)}{\partial z(t)} + \mathcal{O}(\varepsilon^2) \right)}{\varepsilon} \\ &= \lim_{\varepsilon \rightarrow 0^+} \frac{-\varepsilon a(t + \varepsilon) \frac{\partial f(z(t), t, \theta)}{\partial z(t)} + \mathcal{O}(\varepsilon^2)}{\varepsilon} \\ &= \lim_{\varepsilon \rightarrow 0^+} -a(t + \varepsilon) \frac{\partial f(z(t), t, \theta)}{\partial z(t)} + \mathcal{O}(\varepsilon^2) \\ &= -a(t) \frac{\partial f(z(t), t, \theta)}{\partial z(t)} \end{aligned}$$

So its dynamics are governed by another ODE

$$\frac{da(t)}{dt} = -a(t)^T \frac{\partial f(z(t), t, \theta)}{\partial z} \quad (3.2)$$

Finding that adjoint method (3.1) is similar to backpropagation(3.2) because the adjoint state needs to be solved backward in time. Then we specify the constraint on the last time point, the gradient of the loss with respect to the last time point, and obtain the gradient with respect to the hidden state at any time, including the initial value.

By initial condition

$$a(t_N) = \frac{dL}{dz(t_N)} \quad (3.3)$$

$$a(t_0) = a(t_N) + \int_{t_N}^{t_0} \frac{da(t)}{dt} dt = a(t_N) - \int_{t_N}^{t_0} a(t)^T \frac{\partial f(z(t), t, \theta)}{\partial z} dt \quad (3.4)$$

Assume that loss function L depends only on the last point t_N . But if function L depends on intermediate time point, we can repeat the adjoint step for each of the intervals $[t_{N-1}, t_N], [t_{N-2}, t_{N-1}]$ in the backward order and sum up the obtained gradient.

We can generalize (3.2) to obtain gradients with respect to θ , a constant with respect to t , the initial t_0 , and end time t_N . We view θ and t as states with constant differential equations and write

$$\frac{\partial \theta(t)}{\partial t} = 0 \quad \text{and} \quad \frac{dt(t)}{dt} = 1$$

then form an augmented state with corresponding differential equation and adjoint state,

$$\frac{d}{dt} \begin{bmatrix} z \\ \theta \\ t \end{bmatrix} (t) = \begin{bmatrix} f([z, \theta, t]) \\ 0 \\ 1 \end{bmatrix} = f_{aug}([z, \theta, t])$$

$$a_{aug} = \begin{bmatrix} a \\ a_\theta \\ a_t \end{bmatrix} \quad \text{when} \quad a_\theta(t) = \frac{dL}{d\theta(t)} \quad \text{and} \quad a_t(t) = \frac{dL}{dt(t)}$$

consider the Jacobian of f

$$\frac{\partial f_{aug}}{\partial [z, \theta, t]} = \begin{bmatrix} \frac{\partial f}{\partial z} & \frac{\partial f}{\partial \theta} & \frac{\partial f}{\partial t} \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} (t)$$

where each 0 is a matrix of zeros with appropriate dimensions. After that with (3.2)

we obtain

$$\begin{aligned}\frac{da_{aug}(t)}{dt} &= - \begin{bmatrix} a(t) & a_\theta(t) & a_t(t) \end{bmatrix} \frac{\partial f_{aug}}{\partial [z, \theta, t]} \\ &= - \begin{bmatrix} a \frac{\partial f}{\partial z} & a \frac{\partial f}{\partial \theta} & a \frac{\partial f}{\partial t} \end{bmatrix} (t)\end{aligned}$$

Next, setting $a_\theta(t_N) = 0$

consider

$$\begin{aligned}\frac{dL}{d\theta} &= a_\theta(t_0) \\ &= - \int_{t_N}^{t_0} a(t) \frac{\partial f(z(t), t, \theta)}{\partial \theta}\end{aligned}\tag{3.5}$$

and consider gradients with respect to t_0 and t_N

$$\frac{dL}{dt_N} = \frac{dL}{dz(t_N)} \frac{dz(t_N)}{dt_N} = a(t_N) f(z(t_N), t_N, \theta)\tag{3.6}$$

$$\frac{dL}{dt_0} = a_t(t_0) = a_t(t_N) - \int_{t_N}^{t_0} a(t) \frac{\partial f(z(t), t, \theta)}{\partial \theta}\tag{3.7}$$

from (3.2), (3.3), (3.4), (3.5), (3.6), and (3.7) we have gradients for all possible inputs to an initial value problem solver. So NODEs can be composed as the algorithm shown below.

Algorithm 1 : NODEs Algorithm.

Feed-forward

Require : Input values h_0 , activation function σ , and initial weight θ , learning rate η

Define : $dh/dt = \sigma(z, \theta)$

Reverse-mode

Require: $t_0, t_1, \theta, z(t_1)$, loss gradient $\frac{\partial L}{\partial z(t_1)}$

Define initial augmented state:

$$s_0 = [z(t_1), \frac{\partial L}{\partial z(t_1)}, 0_{|\theta|}]$$

Define dynamics on augmented state:

def aug-dynamics($[z(t), a(t), \cdot], t, \theta$)

return $[f(z_t, t, \theta), -a(t)^T \frac{\partial f}{\partial z}, -a(t)^T \frac{\partial f}{\partial \theta}]$

Solve reverse-time ODE:

$$[z(t_0), \frac{\partial L}{\partial z(t_0)}, \frac{\partial L}{\partial \theta}] = \text{ODESolve}(s_0, \text{aug-dynamics}, t_1, t_0, \theta)$$

Return: $\frac{\partial L}{\partial z(t_0)}, \frac{\partial L}{\partial \theta}$

Adjust weight:

$$\theta - = \eta \frac{\partial L}{\partial \theta}$$

3.2 Lookahead Optimizer: k steps forward, 1 step back

Nowadays, many successful deep neural networks employ Stochastic Gradient Descent (SGD) algorithms to training. Recent improvements in SGD can be categorized into two tyoes: adapting learning rate and accelerating the speed of convergence. However, these improvements often required extensive hyperparameter tuning, This is where the Lookahead optimizer comes into play.

Lookahead is orthogonal to these two previous approaches. It maintains a set of "slow weights ϕ " and "fast weights θ ", which get synced with the fast weights every k updates. The fast weights are updated by applying any standard optimization algorithm A , to batches of training examples sampled from the dataset D . After that, the slow weights are updated toward the fast weights by linearly interpolating in weightspace, $\theta - \phi$. We denote the slow weights learning rate as α . After each slow weight has been updated, its current value is considered as the fast weights for the next iteration.

To update the slow weight, we characterize the trajectory as an Exponential Moving Average (EMA) of the final fast weights with in each inner-loop, regard less of the inner optimizer. After k inner-loop steps, we have

$$\begin{aligned}\phi_{t+1} &= \phi_t + \alpha(\theta_{t,k} - \phi_t) \\ &= \alpha[\theta_{t,k} + (1 - \alpha)\theta_{t-1,k} + \dots + (1 - \alpha)^{t-1}\theta_{0,k}] + (1 - \alpha)^t\phi_0\end{aligned}$$

For The fast weights, within each inner-loop, the trajectory of the fast weights depends on the choice of underlying optimizer. Given an optimization algorithm A that takes in an objective function L and the current mini-batch training examples d , the update rule for the fast weights is

$$\theta_{t,i+1} = \theta_{t,i} + A(L, \theta_{t,i-1}, d)$$

Algorithm 2 : Lookahead Optimizer

Require : initial parameter ϕ_0 , slow weight step size α ,

Objective function L , Synchronization period k , Optimizer A

for $t = 1, 2, \dots$

$$\theta_{t,0} = \phi_{t-1}$$

for $i = 1, 2, \dots, k$

sample mini batch of data d

$$\theta_{t,i+1} = \theta_{t,i} + A(L, \theta_{t,i}, d)$$

then

$$\phi_{t+1} = \phi_t + \alpha(\theta_{t,k} - \phi_t)$$

Return : ϕ

Standard optimization methods typically require carefully tuned learning rates to prevent oscillation and slow convergence. Lookahead, however, benefits from a larger learning rate in the inner loop. When oscillating in the high curvature directions, the fast weights updates make rapid progress along the low curvature directions, while the slow weights help smooth out the oscillations through parameter interpolation. The combination of fast weights and slow weights improves learning in high curvature directions, reduces variance, and enables Lookahead to converge rapidly in practice.

3.3 Improving Adjustment of NODEs's Weights

After delving about Neural Ordinary Differential Equations (NODEs) in section 3.1 and Lookahead optimizer in section 3.2, we now pivot the task of integrating these two

algorithms together. The purpose of this integration is to exploit the strengths of NODEs in modeling intricate, continuous-time dynamics along with the robust optimization features offered by the Lookahead algorithm.

Algorithm 3 : NODEs with Lookahead Optimizer.

Feed-forward Algorithm

Require : Input values z_0 , activate function σ , and initial weight θ

Define : $dz/dt = \sigma(z, \theta)$

Reverse-mode algorithm

Require : $t_0, t_1, \theta, z(t_1)$, loss gradient $\frac{\partial L}{\partial z(t_1)}$

Define initial augmented state :

$$s_0 = [z(t_1), \frac{\partial L}{\partial z(t_1)}, 0_{|\theta|}]$$

Define dynamics on augmented state :

def aug-dynamics($[z(t), a(t), \cdot], t, \theta$)

return $[f(z_t, t, \theta), -a(t)^T \frac{\partial f}{\partial z}, -a(t)^T \frac{\partial f}{\partial \theta}]$

Solve reverse-time ODE :

$$[z(t_0), \partial L / \partial z(t_0), \partial L / \partial \theta] = \text{ODESolve}(s_0, \text{aug-dynamics}, t_1, t_0, \theta)$$

Return : $\partial L / \partial z(t_0), \partial L / \partial \theta$

Algorithm 4 : Adjust Weight Algorithm

Require : initial parameter ϕ_0 , slow weight step size α ,

Objective function L , Synchronization period k , Optimizer A

initialization:

$$\theta_{t,0} = \phi_{t-1}$$

Forward Updates:

for $i = 1, 2, \dots, k$

sample mini batch of data d

$$\theta_{t,i+1} = \theta_{t,i} + A(L, \theta_{t,i}, d)$$

Update Slow Weights:

$$\phi_{t+1} = \phi_t + \alpha(\theta_{t,k} - \phi_t)$$

Return : ϕ

Chapter 4

Experimental Study

The primary aim of this chapter is to conduct an experimental study on the training of Neural ODEs with fixed initial and target states. For the purpose of this experimental study, we consider a simple test scenario with fixed initial and target states

Initial State $z_0 = (1, 2)$

Target State $= (2, 4)$

In The course of this experimental study, we focused on the following

1. Implementing a Neural ODE with a sigmoid activation function.
2. Utilizing the Glorot (Xavier) weight initialization set the initial weights.
3. Fixed Parameters
 - Time Interval (dt): Set at 0.2.
 - Maximum Time (t_max): Set at 2.
 - Stopping Criterion (stop_loss_diff): Algorithm halts if the difference between consecutive loss values falls below 0.0007.
4. Hyperparameters:
 - Learning Rate (lr): 0.01, 0.05 and 0.1.
 - Lookahead Steps (k_lookahead): 4, 6 and 8.
5. Using the Euler forward method for solving the Neural ODE.
6. Investigating the impact of different optimization techniques, namely standard gradient descent and the Lookahead optimizer, on the training process.

The loss function is defined as the squared Euclidean distance between the final state and the target state, with the objective being to minimize this loss through optimization.

Forward Pass

Source Code

Listing4.1: Forward pass

```
1      def sigmoid(x):
2          return 1 / (1 + np.exp(-x))
3
4      def neural_ode_forward(z, t, W, b):
5          dzdt = sigmoid(np.dot(W, z) + b)
6          return dzdt
7
8      def euler_forward(z0, t, W, b, dt):
9          z = z0
10         z_history = [z0]
11         for _ in t[1:]:
12             z = z + dt * neural_ode_forward(z, _, W, b)
13             z_history.append(z)
14         return z, np.array(z_history)
15
```

The forward pass of the neural ODE is governed by the ordinary differential equation

$$\frac{dz}{dt} = \sigma(Wz + b)$$

Here, $\sigma(x)$ is the sigmoid activation function, W is the weight matrix, b is the bias vector, and z is the state vector.

Using Euler's method, the forward pass can be approximated as

$$z_{t+1} = z_t + \Delta t \cdot \sigma(Wz_t + b)$$

for example with hyperparameters

$$z_0 = \begin{bmatrix} 1 & 1 \end{bmatrix}, \quad \text{Target} = \begin{bmatrix} 2 & 1 \end{bmatrix}$$

$$w = \begin{bmatrix} 1 & 0 \\ 0 & 2 \end{bmatrix}, \quad b = \begin{bmatrix} 1 & 1 \end{bmatrix} \quad \text{and} \quad dt = 0.5$$

Then the forward pass process as

$$\begin{aligned} z(0.5) &= z(0) + dt \left(\frac{dz}{dt} \right) \\ &= z(0) + dt \sigma(w \cdot z(0) + b) \\ &= \begin{bmatrix} 1 & 1 \end{bmatrix} + 0.5\sigma \begin{bmatrix} 2 & 3 \end{bmatrix} \\ &= \begin{bmatrix} 1.44 & 1.48 \end{bmatrix} \\ z(1) &= z(0.5) + 0.5\sigma \begin{bmatrix} 2.44 & 3.96 \end{bmatrix} \\ &= \begin{bmatrix} 1.9 & 1.97 \end{bmatrix} \\ z(1.5) &= z(1) + 0.5\sigma \begin{bmatrix} 2.9 & 4.94 \end{bmatrix} \\ &= \begin{bmatrix} 2.37 & 2.47 \end{bmatrix} \\ &\vdots \end{aligned}$$

Backward Pass (Adjoint Sensitivity)

Source Code

Listing4.2: Backward pass

```
1      def euler_backward(z_history, grad_output, W, b, dt):
2          adjoint = grad_output
3          grad_W = np.zeros_like(W)
4          grad_b = np.zeros_like(b)
5          for z in reversed(z_history[:-1]):
6              grad_W += dt * np.outer(adjoint, z)
7              grad_b += dt * adjoint
8              adjoint = adjoint - dt * np.dot(W.T, adjoint *
          sigmoid(z) *
9              (1 - sigmoid(z)))
10         return grad_W, grad_b
11
```

The backward pass aims to compute the gradients $\frac{\partial L}{\partial W}$ and $\frac{\partial L}{\partial b}$, where L is the loss function.

The adjoint sensitivity method introduces an adjoint state a , which is the gradient of the loss with respect to the state z

$$a = \frac{\partial L}{\partial z}$$

The adjoint equation is

$$\frac{da}{dt} = -(W^T a \odot \sigma'(Wz_t + b))$$

Here, \odot denotes element-wise multiplication, and $\sigma'(x)$ is the derivative of the sigmoid function.

Using Euler's method, the adjoint equation can be approximated as

$$a_{t-1} = a_t - \Delta t \cdot (W^T a_t) \odot \sigma'(Wz_t + b)$$

The gradients $\frac{\partial L}{\partial W}$ and $\frac{\partial L}{\partial b}$ can be updated as

$$\begin{aligned}\frac{\partial L}{\partial W} &+ = \Delta t \cdot a_t \otimes z_t \\ \frac{\partial L}{\partial b} &+ = \Delta t \cdot a_t\end{aligned}$$

Here, \otimes denotes the outer product.

Lookahead Optimizer

Source Code

Listing4.3: Lookahead Optimizer

```
1      class Lookahead:
2          def __init__(self, alpha=0.5, k=5):
3              self.alpha = alpha
4              self.k = k
5              self.count = 0
6              self.slow_weights_W = None
7              self.slow_weights_b = None
8
9          def step(self, W, b, grad_W, grad_b, lr):
10             if self.slow_weights_W is None:
11                 self.slow_weights_W, self.slow_weights_b =
12                     W.copy(), b.copy()
13             W -= lr * grad_W
14             b -= lr * grad_b
15             self.count += 1
16             if self.count % self.k == 0:
17                 self.slow_weights_W +=
18                     self.alpha * (W - self.slow_weights_W)
19                 self.slow_weights_b +=
20                     self.alpha * (b - self.slow_weights_b)
21                 W, b =
22                     self.slow_weights_W.copy(), self.slow_weights_b.copy
23
24             return W, b
```

The Lookahead optimizer involves two sets of weights, the "fast weights" W_f , b_f and the "slow weights" W_s , b_s . The fast weights are updated more frequently, while the slow weights are updated less often but guide the overall optimization.

The update rules for Lookahead are

$$\begin{aligned} W_f &= W_f - \eta \frac{\partial L}{\partial W} \\ b_f &= b_f - \eta \frac{\partial L}{\partial b} \end{aligned}$$

Every k steps, the slow weights are updated as

$$\begin{aligned} W_s &= W_s + \alpha(W_f - W_s) \\ b_s &= b_s + \alpha(b_f - b_s) \end{aligned}$$

And then the fast weights are set to the slow weights

$$\begin{aligned} W_f &= W_s \\ b_f &= b_s \end{aligned}$$

Here, η is the learning rate, and α is the interpolation parameter for Lookahead.

Experimental Setup and Results

To evaluate the stability and efficiency of these optimization strategies of these methods, we varied two key hyperparameters

- **Lookahead Steps (k)**

We tested three different values for k in the Lookahead optimizer $k = 4, 6$ and 8

- **Learning Rate**

For each value of k , we experimented with four different learning rates $\text{lr} = 0.01, 0.05$ and 0.1

- **Data Collection**

The loss function, defined as the squared Euclidean distance between the final and target states, was recorded at each iteration for all combinations of k and learning rate.

- **Results and Figures**

The results are visualized through a series of graphs that plot the loss function against the number of iterations. These graphs provide insights into the convergence behavior and stability of each optimization method under different hyperparameters.

when $k = 4$ we have

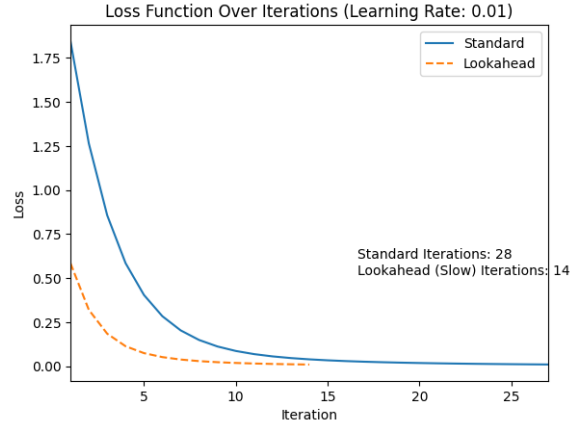


Figure 4 : Illustrates the loss function when $k = 4$ and $lr = 0.01$.

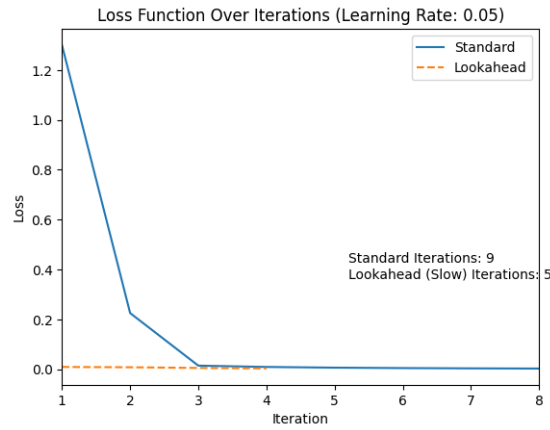


Figure 5 : Illustrates the loss function when $k = 4$ and $lr = 0.05$.

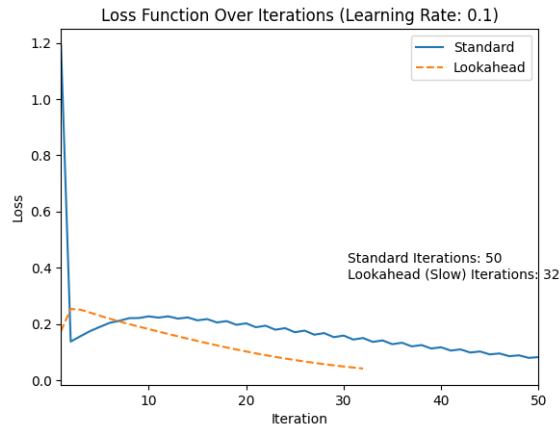


Figure 6 : Illustrates the loss function when $k = 4$ and $lr = 0.1$.

when $k = 6$ we have

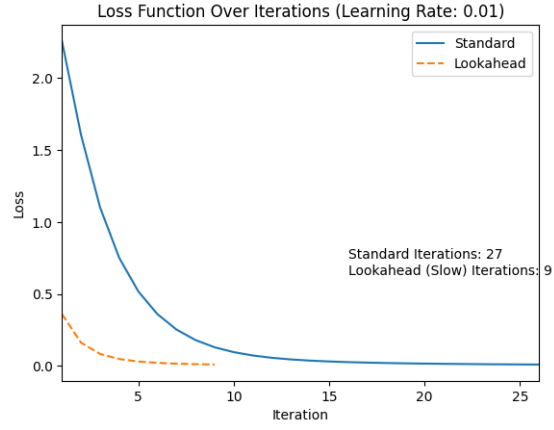


Figure 7 : Illustrates the loss function when $k = 6$ and $lr = 0.01$.

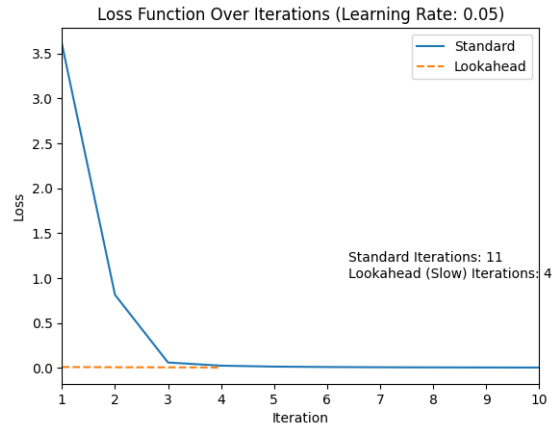


Figure 8 : Illustrates the loss function when $k = 6$ and $lr = 0.05$.

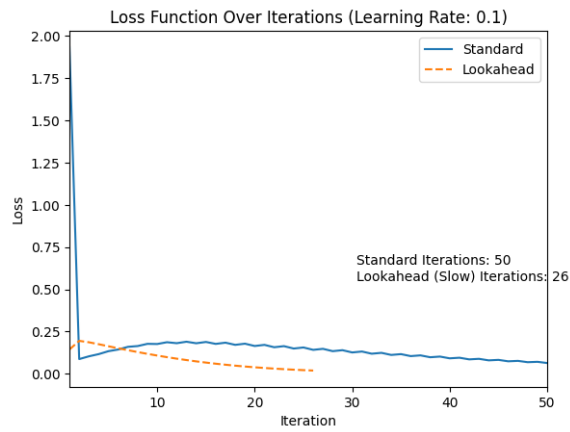


Figure 9 : Illustrates the loss function when $k = 6$ and $lr = 0.1$.

when $k = 8$ we have

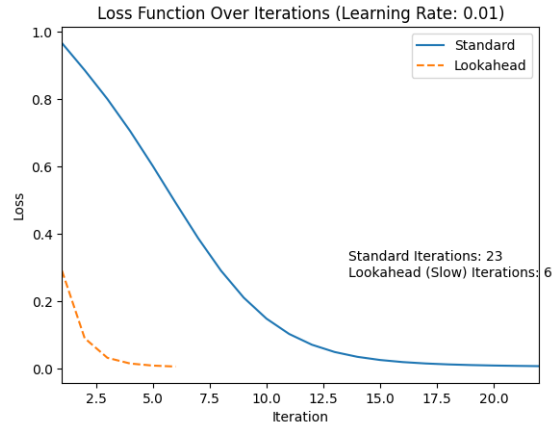


Figure 10 : Illustrates the loss function when $k = 8$ and $lr = 0.01$.

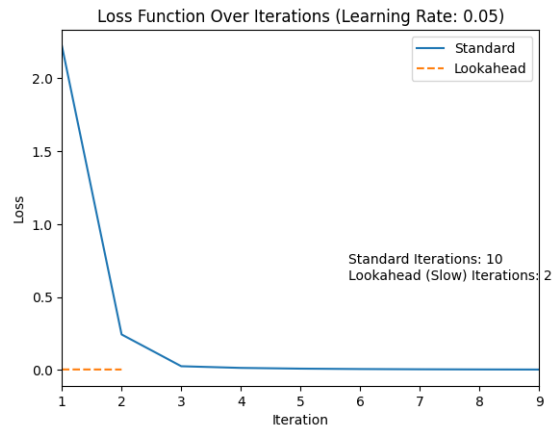


Figure 11 : Illustrates the loss function when $k = 8$ and $lr = 0.05$.

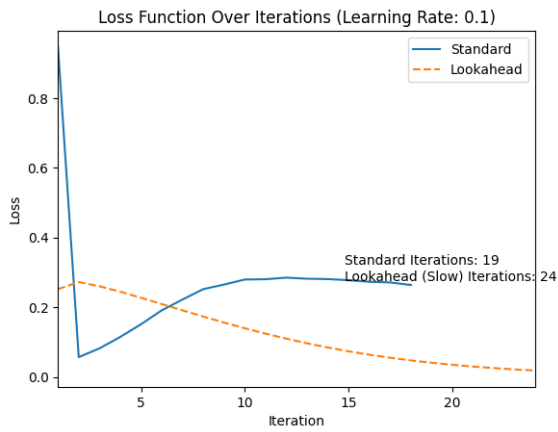


Figure 12 : Illustrates the loss function when $k = 8$ and $lr = 0.1$.

Our experimental evaluation focused on the performance of Neural ODEs under two optimization strategies: Standard Gradient Descent and the Lookahead Optimizer. We investigated three different scenarios for each optimization method as depicted in Figures 4-12.

for all tested learning rate and all tested values of k in both optimization methods, the method converged as can be seen in Figure 4-6 for $k = 4$, Figure 7-9 for $k = 6$, and Figure 10-12 for $k = 8$. However, the Lookahead optimizer, in particular, showed faster convergence compared to Standard Gradient Descent, a trend that is evident from the fewer number of iterations required for convergence in the aforementioned figures. This suggests that the Lookahead optimizer’s ability to focus on long-term updates provides a significant advantage in terms of convergence speed.

Moreover, while the Lookahead optimizer exhibited a smooth convergence, the Standard Gradient Descent occasionally showed oscillatory behaviors, especially at higher learning rates. This suggests that the Lookahead optimizer’s ability to focus on long-term updates provides a significant advantage in terms of convergence speed and stability.

Summary of Results

Our experiments, supported by the results presented in Figure 4-12, provide empirical evidence of the effectiveness of the Lookahead optimizer in the context of Neural ODEs. Specifically, the Lookahead optimizer demonstrates in

1. Accelerating convergence at lower learning rates.
2. Offering greater initial stability at higher learning rates where Standard Gradient Descent tends to oscillate.

These findings contribute valuable insights into the optimization landscape of Neural ODEs and set the stage for future research in this area.

Chapter 5

Conclusion

Our experimental study has yielded valuable insights into the optimization of Neural Ordinary Differential Equations (Neural ODEs) through the Lookahead Optimizer. Specifically, we found that the Lookahead Optimizer facilitates faster convergence at lower learning rates and provides greater initial stability at higher learning rates [9]. These findings contribute to the growing body of empirical evidence supporting the effectiveness of Lookahead Optimizers in the context of Neural ODEs [2, 9].

Beyond the immediate results, the implications of using the Lookahead Optimizer with Neural ODEs are manifold. The Lookahead Optimizer’s robustness to higher learning rates makes it a particularly strong candidate for Neural ODEs, which often exhibit sensitivity to hyperparameter settings [2]. This robustness could translate to computational efficiency, as faster convergence would reduce the number of required iterations. Furthermore, the Lookahead Optimizer’s flexibility across a range of learning rates suggests its potential as a versatile optimization strategy for Neural ODEs, which could be particularly beneficial for complex or large-scale problems [9]. Given the widespread application of ODEs in various scientific disciplines, these benefits could extend to real-world problems in physics, biology, and engineering [12].

The promising results of this study open several avenues for future research. One immediate direction is the formal mathematical investigation of the Lookahead Optimizer’s properties, particularly its convergence and stability when applied to Neural ODEs [2]. Additionally, the use of more advanced ODE solvers like Runge-Kutta methods could offer improvements in numerical stability and accuracy [12]. The exploration of adaptive learning rate methods such as Stochastic Gradient Descent (SGD) or Adam could also yield more nu-

anced insights into the optimization landscape of Neural ODEs [2, 9]. Finally, extending this work to include more complex Neural ODE architectures or different activation functions could provide a comprehensive understanding of the Lookahead Optimizer’s effectiveness and limitations [2].

References

- [1] Coddington, E. A., & Levinson, N. (1955). Theory of Ordinary Differential Equations. McGraw-Hill.
- [2] Chen, R. T. Q., Rubanova, Y., Bettencourt, J., & Duvenaud, D. K. (2018). Neural Ordinary Differential Equations. In Advances in Neural Information Processing Systems.
- [3] Kidger, P., Lyons, T., & Morrill, J. (2020). Universal Differential Equations for Scientific Machine Learning. arXiv preprint arXiv:2001.04385.
- [4] Zhang, J., Mitliagkas, I., & Rabbat, M. (2019). YellowFin and the Art of Tuning Hyperparameters. arXiv preprint arXiv:1706.03471.
- [5] Glorot, X., & Bengio, Y., (2010), Understanding the difficulty of training deep feed-forward neural networks. In Proceedings of the thirteenth international conference on artificial intelligence and statistics (pp. 249-256).
- [6] Kingma, D. P., & Ba, J. (2014). Adam: A Method for Stochastic Optimization. arXiv preprint arXiv:1412.6980.
- [7] Cheney W. (2001), "The Chain Rule and Mean Value Theorems", Analysis for Applied Mathematics, New York: Springer, page 121125.
- [8] Gavril Ognjanovski (2019), Toward Data Science, accessed 21 september 2023, <<https://towardsdatascience.com/everything-you-need-to-know-about-neural-networks-and-backpropagation-machine-learning-made-easy-e5285bc2be3a>>
- [9] Zhang, M., Lucas, J., Hinton, G., & Ba, J. (2019). Lookahead Optimizer: k steps forward, 1 step back. arXiv preprint arXiv:1907.08610.
- [10] Butcher, J. C. (2008). Numerical Methods for Ordinary Differential Equations. John Wiley & Sons.

- [11] Ruthotto, L., & Haber, E. (2018). Deep Neural Networks motivated by Partial Differential Equations. arXiv preprint arXiv:1804.04272.
- [12] Hairer, E., Nørsett, S. P., & Wanner, G. (1993). Solving Ordinary Differential Equations I: Nonstiff Problems. Springer-Verlag.
- [13] Kaiming H., Xiangyu Zh., Shaoqing R., & Jian S. (2015), Deep Residual Learning for Image Recognition, - arXiv preprint, <arXiv:1512.00567>.
- [14] S. Ruder, "An overview of gradient descent", NUI galway Aylien Ltd., Dublin, - arXiv preprint, <arXiv:1609.04747>, 2016

Appendix

Listing 5.1: Code

```
1      import numpy as np
2      import matplotlib.pyplot as plt
3
4      def sigmoid(x):
5          return 1 / (1 + np.exp(-x))
6
7      def neural_ode_forward(z, t, W, b):
8          dzdt = sigmoid(np.dot(W, z) + b)
9      return dzdt
10
11     def euler_forward(z0, t, W, b, dt):
12         z = z0
13         z_history = [z0]
14         for _ in t[1:]:
15             z = z + dt * neural_ode_forward(z, _, W, b)
16             z_history.append(z)
17         return z, np.array(z_history)
18
19     def euler_backward(z_history, grad_output, W, b, dt):
20         adjoint = grad_output
21         grad_W = np.zeros_like(W)
22         grad_b = np.zeros_like(b)
23         for z in reversed(z_history[:-1]):
24             grad_W += dt * np.outer(adjoint, z)
25             grad_b += dt * adjoint
26             adjoint = adjoint - dt * np.dot(W.T, adjoint * sigmoid(z) *
```

```

27         (1 - sigmoid(z)))
28     return grad_W, grad_b
29
30     class Lookahead:
31         def __init__(self, alpha=0.5, k=5):
32             self.alpha = alpha
33             self.k = k
34             self.count = 0
35             self.slow_weights_W = None
36             self.slow_weights_b = None
37
38         def step(self, W, b, grad_W, grad_b, lr):
39             if self.slow_weights_W is None:
40                 self.slow_weights_W, self.slow_weights_b =
41                     W.copy(), b.copy()
42             W -= lr * grad_W
43             b -= lr * grad_b
44             self.count += 1
45             if self.count % self.k == 0:
46                 self.slow_weights_W +=
47                     self.alpha * (W - self.slow_weights_W)
48                 self.slow_weights_b +=
49                     self.alpha * (b - self.slow_weights_b)
50             W, b =
51                 self.slow_weights_W.copy(), self.slow_weights_b.copy()
52         return W, b
53
54     # Xavier/Glorot Initialization
55     n_input = 2 # Number of input features
56     n_output = 2 # Number of output features
57     initial_W = np.random.randn(n_input, n_output) *
58         np.sqrt(2. / (n_input + n_output))
59     initial_b = np.random.randn(n_output) *
60         np.sqrt(2. / (n_input + n_output))
61
62     # Copy initial parameters for both training methods

```

```

63     W_standard = initial_W.copy()
64     b_standard = initial_b.copy()
65
66     W_lookahead = initial_W.copy()
67     b_lookahead = initial_b.copy()
68
69     # Hyperparameters
70     lr = 0.01
71     dt = 0.2
72     t_max=2
73     t = np.linspace(0, t_max, int(t_max/dt) + 1)
74     iterations_standard = 50
75     k_lookahead = 5
76     iterations_lookahead = iterations_standard * k_lookahead
77     stop_loss_diff = 0.0007
78
79     # Initialize variables to keep track of previous loss
80     prev_loss_standard = None
81     prev_loss_lookahead = None
82
83     # Training data (dummy)
84     z0 = np.array([1.0, 2.0])
85     target = np.array([2.0, 4.0])
86
87     # For standard training
88     losses_standard = []
89
90     # For Lookahead
91     lookahead = Lookahead(alpha=0.5, k=k_lookahead)
92     losses_lookahead = []
93
94     # For Lookahead with slow weight updates only at multiples of k
95     losses_lookahead_slow = []
96
97     # Standard training loop
98     for k in range(1, iterations_standard + 1):

```

```

99         z, z_history = euler_forward(z0, t, W_standard, b_standard, dt)
100         loss = np.sum((z - target)**2)
101         if prev_loss_standard is not None:
102             loss_diff = prev_loss_standard - loss
103             if 0 < loss_diff < stop_loss_diff:
104                 print(f"Stopping criterion met at iteration {k} (Standard
105 ). Loss Difference: {loss_diff}")
106                 break
107             prev_loss_standard = loss
108             grad_output = 2 * (z - target)
109             grad_W, grad_b = euler_backward(z_history, grad_output,
110 W_standard, b_standard, dt)
111             W_standard -= lr * grad_W
112             b_standard -= lr * grad_b
113             losses_standard.append(loss)
114             print(f"Iteration {k}
115 (Standard), Loss: {loss}")
116         print(f"Total number of iterations (Standard): {k}")
117         total_iterations_standard = k
118
119     # Lookahead training loop
120     for k in range(1, iterations_lookahead + 1):
121         z, z_history = euler_forward(z0, t, W_lookahead, b_lookahead, dt)
122         loss = np.sum((z - target)**2)
123         if prev_loss_lookahead is not None:
124             loss_diff = prev_loss_lookahead - loss
125             if 0 < loss_diff < stop_loss_diff:
126                 print(f"Stopping criterion met at iteration {k} (
127 Lookahead). Loss Difference: {loss_diff}")
128                 break
129             prev_loss_lookahead = loss
130             grad_output = 2 * (z - target)
131             grad_W, grad_b = euler_backward(z_history, grad_output,
132 W_lookahead, b_lookahead, dt)
133             W_lookahead, b_lookahead = lookahead.step(W_lookahead,
134 b_lookahead, grad_W, grad_b, lr)

```

```

130         if k <= 100:
131             losses_lookahead.append(loss)
132         if k % lookahead.k == 0:
133             losses_lookahead_slow.append(loss)
134         if k <= 100:
135             print(f"Iteration {k} (Lookahead), Loss: {loss}")
136             print(f"Total number of iterations (Lookahead): {k}")
137             print(f"Total number of slow weight updates (Lookahead): {k //
lookahead.k}")
138         total_iterations_lookahead = k
139         total_iterations_lookahead_slow = k // lookahead.k
140
141         # Plotting the loss
142         plt.figure()
143         plt.plot(range(1, len(losses_standard)+ 1), losses_standard, label='
Standard')
144         # plt.plot(range(1, len(losses_lookahead) + 1), losses_lookahead,
label='Lookahead')
145         plt.plot(range(1, len(losses_lookahead_slow) + 1),
losses_lookahead_slow, label='Lookahead', linestyle='--')
146         plt.xlabel('Iteration')
147         plt.ylabel('Loss')
148         plt.title(f'Loss Function Over Iterations (Learning Rate: {lr})')
149         plt.xlim(1, max(len(losses_standard), len(losses_lookahead_slow)))
150
151         # Annotate the number of iterations for each method
152         annotation_text = f"Standard Iterations: {total_iterations_standard}\
n"
153         # annotation_text += f"Lookahead Iterations: {
total_iterations_lookahead}\n"
154         annotation_text += f"Lookahead (Slow) Iterations: {
total_iterations_lookahead_slow}"
155
156         plt.annotate(annotation_text, xy=(0.6, 0.3), xycoords='axes fraction'
)
157

```



```
158     plt.legend()
159     plt.show()
160
161     # Plotting the forward Euler (Last iteration)
162     plt.figure()
163     plt.plot(t, np.array(z_history)[:, 0], label='x-component')
164     plt.plot(t, np.array(z_history)[:, 1], label='y-component', linestyle
165 = '--')
166     plt.xlabel('Time')
167     plt.ylabel('State Value')
168     plt.title('Forward Euler State Evolution (Last Iteration)')
169     plt.legend()
170     plt.show()
```


OPTIMIZING NEURAL ORDINARY DIFFERENTIAL EQUATIONS WITH LOOKAHEAD OPTIMIZER

NIYATA SANNGAI STUDENT ID : 630510482

ADVISOR : ASST. PROF. DR.NUTTAWAT SONTICHA

DEPARTMENT OF MATHEMATICS, FACULTY OF SCIENCE, CHIANG MAI UNIVERSITY

ABSTRACT

THIS STUDY DELVES INTO THE APPLICATION OF NEURAL ORDINARY DIFFERENTIAL EQUATIONS (NODES) WITHIN THE MACHINE LEARNING DOMAIN, INTRODUCING AN INNOVATIVE OPTIMIZATION STRATEGY USING THE LOOKAHEAD OPTIMIZER. BY UTILIZING EULER'S METHOD FOR BOTH FORWARD AND ADJOINT SENSITIVITY CALCULATIONS. OUR EXPERIMENTAL RESULTS INDICATE THAT THE LOOKAHEAD OPTIMIZER CONVERGES FASTER AT LOWER LEARNING RATES AND EXHIBITS GREATER STABILITY AT HIGHER LEARNING RATES COMPARED TO STANDARD GRADIENT DESCENT. THIS WORK PROVIDES EMPIRICAL EVIDENCE SUPPORTING THE EFFICACY OF THE LOOKAHEAD OPTIMIZER IN NEURAL ODE CONTEXTS AND OFFERS AN OPEN-SOURCE CODEBASE FOR FUTURE RESEARCH.

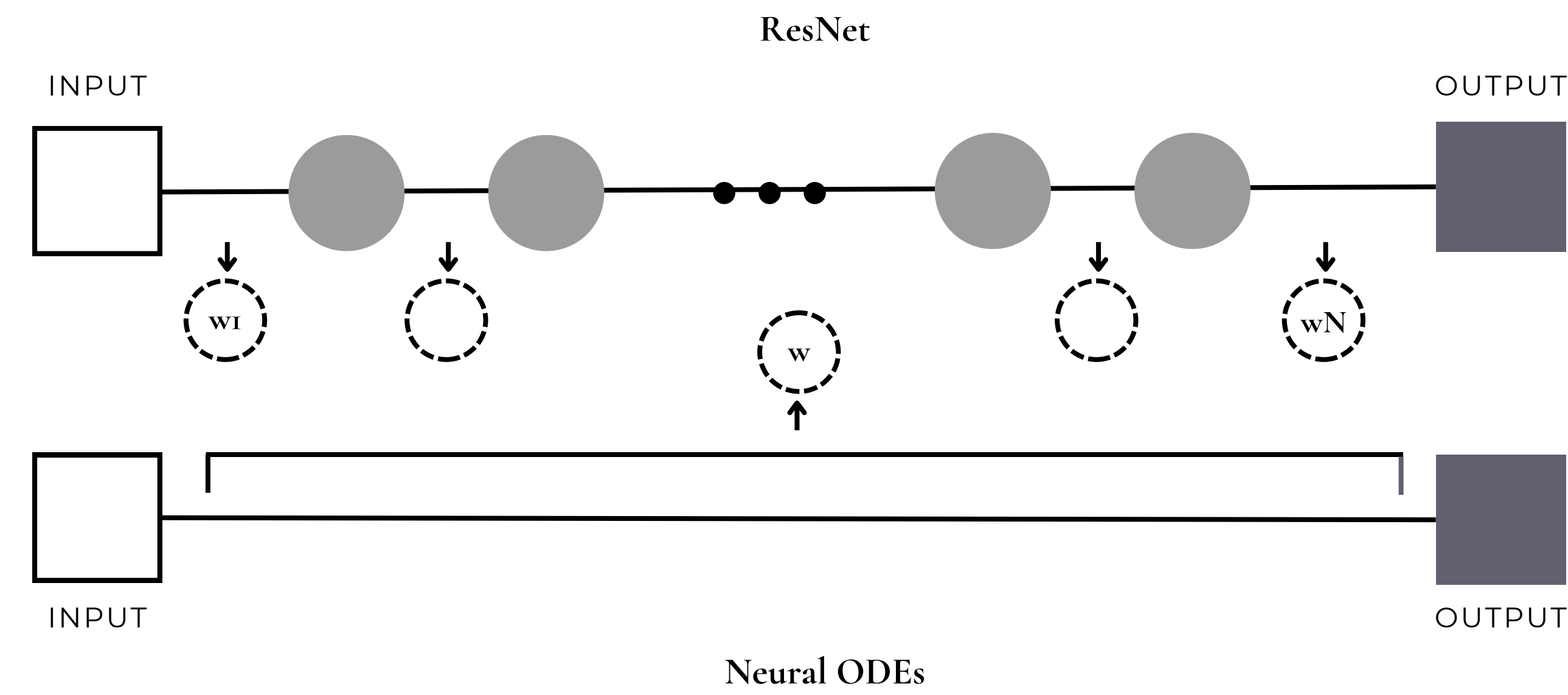
METHODOLOGY

• NEURAL ORDINARY DIFFERENTIAL EQUATIONS (NODES)

NODES OFFER A NEW FAMILY OF DEEP NEURAL NETWORK ARCHITECTURES THAT HAVE THE PERSPECTIVE TO SEAMLESSLY COMPUTE INTERMEDIATE DERIVATIVES OF FLOW OF HIDDEN LAYER STATES, CREATING CONTINUOUS-DEPTH MODELS BY LET

$$\frac{dh_t}{dt} = f(h_t, t, \theta)$$

THIS METHOD ALLOWS US TO TRAIN MODELS BY USING NODES WHICH CAN BE CONSIDERED AS APPROXIMATIONS BASED ON ONE-STEP DISCRETIZATION OF CONTINUOUS DYNAMICS.



FOR BACK PROPAGATION THEY USING THE ADJOINT SENSITIVITY. THIS METHOD IS APPLICABLE TO ALL ODE SOLVERS, SCALES LINEARLY WITH PROBLEM SIZE, HAS LOW MEMORY COSTS, AND CONTROLS NUMERICAL ERRORS EFFECTIVELY.

THE GRADIENT OF THE LOSS WITH RESPECT TO HIDDEN STATE Z(T) AT EACH TIME IS DEFINED AS THE ADJOINT

$$a(t) = \frac{dL}{dz(t)}$$

SO WE CAN COMPUTE A GRADIENT OF LOSS WITH RESPECT TO WEIGHT TO

$$\frac{dL}{d\theta} = - \int_{t_N}^{t_0} a(t) \frac{\partial f(z(t), t, \theta)}{\partial \theta}$$

ADJOINT SENSITIVITY METHOD CAN BE WRITTEN IN ALGORITHM BELOW

Algorithm Reverse-mode derivative of an ODE initial value problem

Input: dynamics parameters θ , start time t_0 , stop time t_1 , final state $\mathbf{z}(t_1)$, loss gradient $\partial L / \partial \mathbf{z}(t_1)$

$s_0 = [\mathbf{z}(t_1), \frac{\partial L}{\partial \mathbf{z}(t_1)}, \mathbf{0}_{|\theta|}]$ ▷ Define initial augmented state

def aug_dynamics($[\mathbf{z}(t), \mathbf{a}(t), \cdot], t, \theta$):

▷ Define dynamics on augmented state

return $[f(\mathbf{z}(t), t, \theta), -\mathbf{a}(t)^T \frac{\partial f}{\partial \mathbf{z}}, -\mathbf{a}(t)^T \frac{\partial f}{\partial \theta}]$ ▷ Compute vector-Jacobian products

$[\mathbf{z}(t_0), \frac{\partial L}{\partial \mathbf{z}(t_0)}, \frac{\partial L}{\partial \theta}] = \text{ODESolve}(s_0, \text{aug_dynamics}, t_1, t_0, \theta)$ ▷ Solve reverse-time ODE

return $\frac{\partial L}{\partial \mathbf{z}(t_0)}, \frac{\partial L}{\partial \theta}$ ▷ Return gradients

• LOOKAHEAD OPTIMIZER: K STEPS FORWARD, 1 STEP BACK

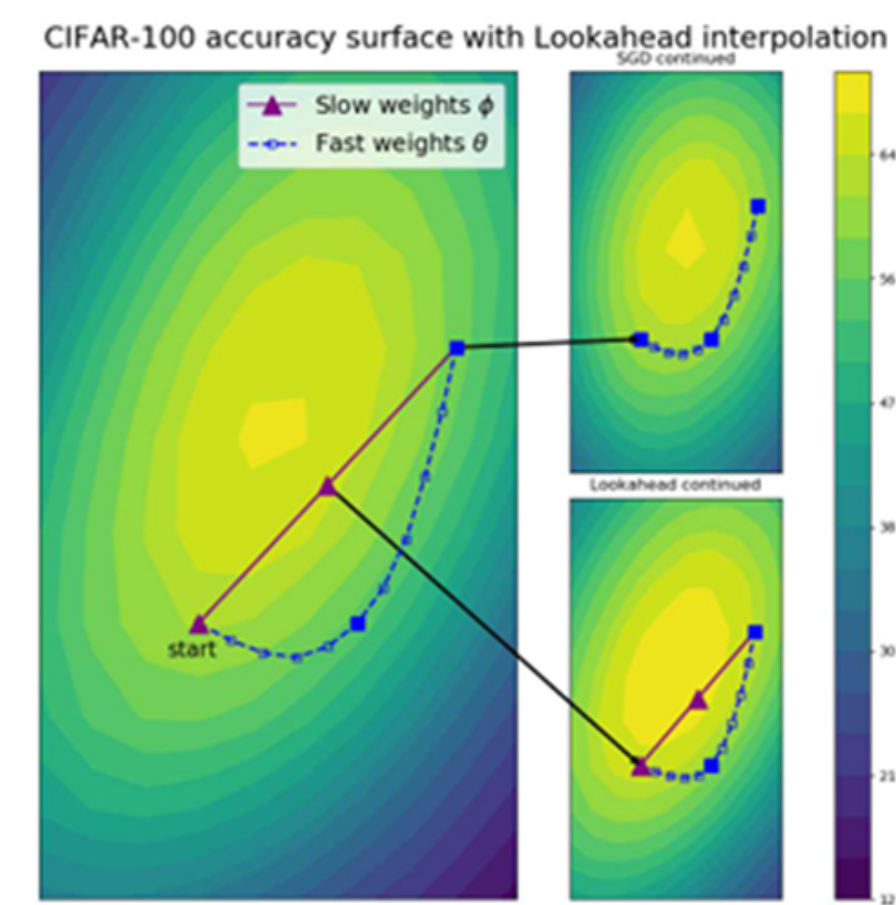
THE LOOKAHEAD OPTIMIZER ADAPTS LEARNING RATES TO IMPROVE CONVERGENCE SPEED. IT MAINTAINS A SET OF SLOW WEIGHTS AND FAST WEIGHTS, WITH THE FORMER SERVING AS A REFERENCE FOR THE LATTER, WHICH GET SYNCED WITH THE FAST WEIGHTS EVERY K UPDATES.

$$\theta_{t,i+1} = \theta_{t,i} + A(L, \theta_{t,i-1}, d)$$

OPTIMIZATION THAT ALIGNS AND UPDATES WEIGHTS BY LINEARLY INTERPOLATING TOWARD THE FAST WEIGHTS.

$$\phi_{t+1} = \phi_t + \alpha(\theta_{t,k} - \phi_t)$$

ONCE THE FAST WEIGHTS ARE UPDATED, THEIR CURRENT VALUE IS CONSIDERED AS THE REFERENCE FOR THE NEXT ITERATION.



Algorithm 1 Lookahead Optimizer:

Require: Initial parameters ϕ_0 , objective function L

Require: Synchronization period k , slow weights step size α , optimizer A

for $t = 1, 2, \dots$ **do**

Synchronize parameters $\theta_{t,0} \leftarrow \phi_{t-1}$

for $i = 1, 2, \dots, k$ **do**

sample minibatch of data $d \sim \mathcal{D}$

$\theta_{t,i} \leftarrow \theta_{t,i-1} + A(L, \theta_{t,i-1}, d)$

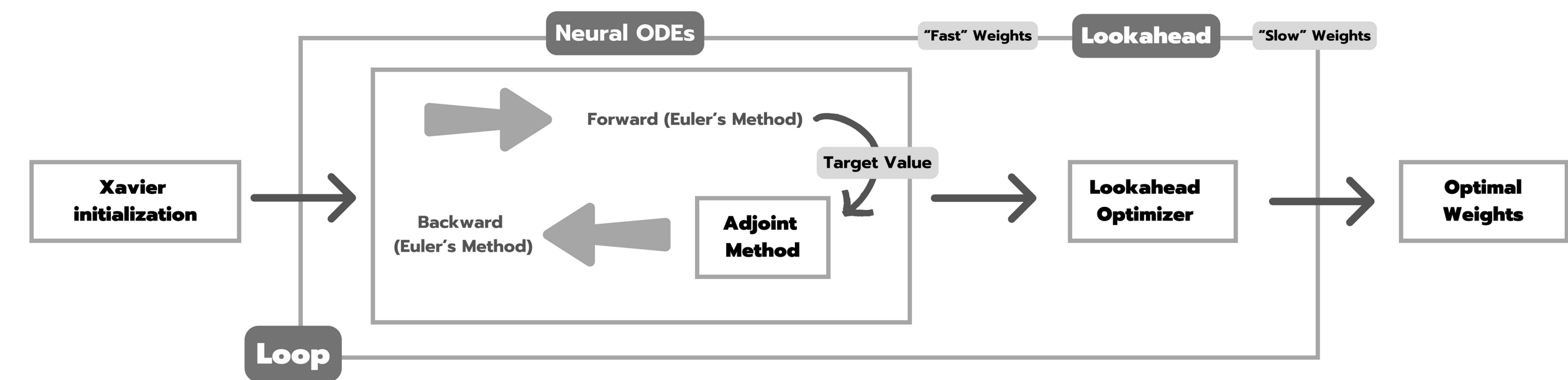
end for

Perform outer update $\phi_t \leftarrow \phi_{t-1} + \alpha(\theta_{t,k} - \phi_{t-1})$

end for

return parameters ϕ

• NEURAL ODE OPTIMIZATION WORKFLOW WITH LOOKAHEAD OPTIMIZER



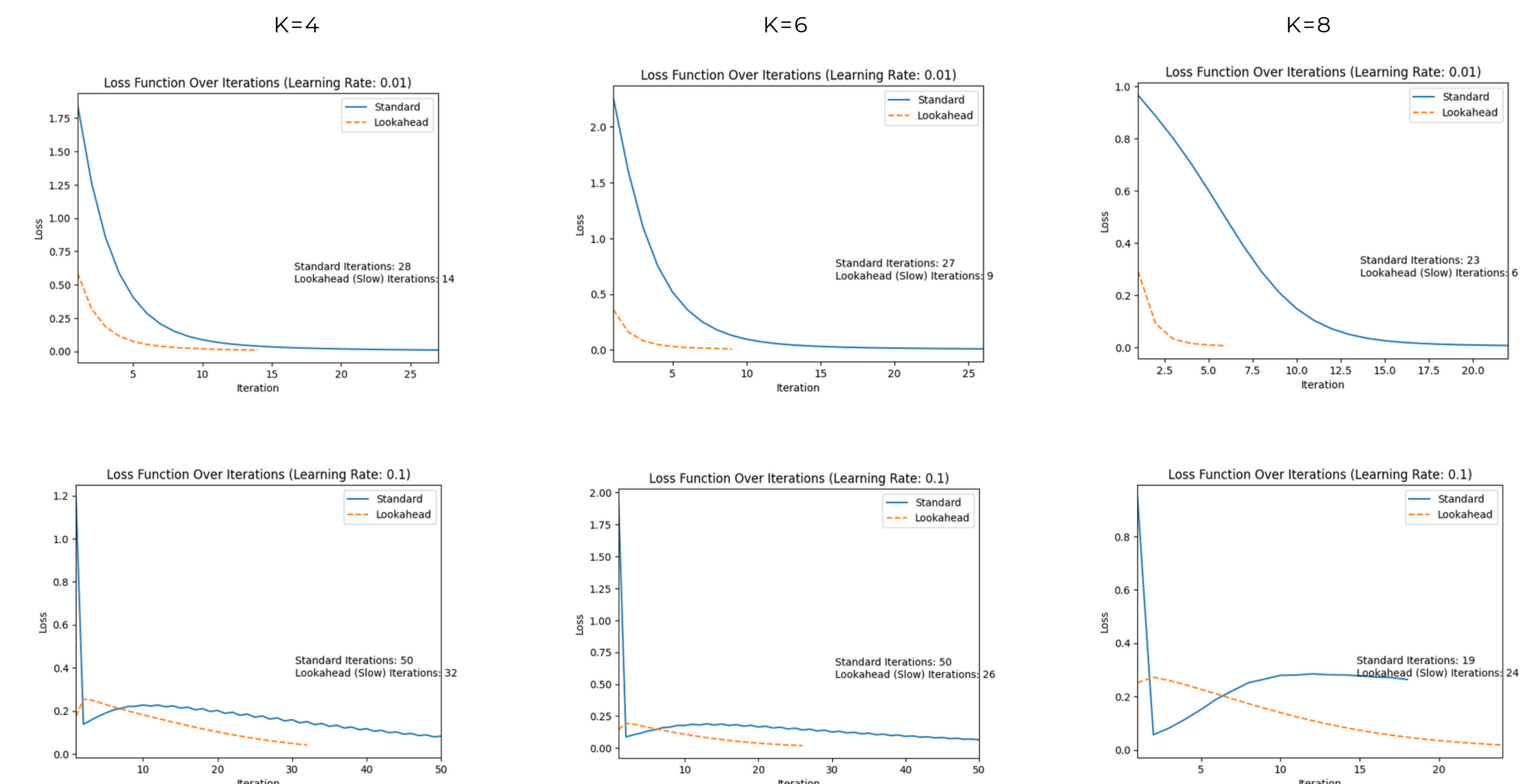
EXPERIMENTAL STUDY

THE PRIMARY AIM OF THIS CHAPTER IS TO CONDUCT AN EXPERIMENTAL STUDY ON THE TRAINING OF NEURAL ODES. WE CONSIDERED A SIMPLE TEST SCENARIO WITH FIXED HYPERPARAMETERS, INITIAL STATES, AND TARGET STATES WHICH ARE PRESENTED BELOW

- INITIAL STATES : (1,2)
- TARGET STATES : (2,4)
- LEARNING RATES : 0.01, 0.05, 0.1
- LOOKAHEAD STEPS (K) : 4, 6, 8

MOREOVER WE USING THE XAVIER INITIALIZATION METHOD, SHOWN BELOW, TO SET UP THE INITIAL WEIGHTS.

RESULTS



CONCLUSION

THE LOOKAHEAD OPTIMIZER EXHIBITS QUICKER CONVERGENCE THAN STANDARD GRADIENT DESCENT. THIS EFFICIENCY SUGGESTS THAT FEWER ITERATIONS ARE NEEDED FOR CONVERGENCE. THE LOOKAHEAD OPTIMIZER'S ADAPTIVE UPDATE MECHANISM OFFERS ADVANTAGES IN TERMS OF SPEED AND STABILITY, ESPECIALLY AT HIGHER LEARNING RATES.

ACKNOWLEDGEMENTS

THIS WORK WAS SUPPORTED BY A SCHOLARSHIP FROM THE DEVELOPMENT AND PROMOTION OF SCIENCE AND TECHNOLOGY TALENTS PROJECT (DPST).

References



Source code

