University College Cork

Department of Computer Science

Optimal Control and Backpropagation

Suneet Mahajan

Supervisor: Michel Schellekens

Analytics Project for Computer Science

April 17, 2025

Declaration

In signing this declaration, you are conforming, in writing, that the submitted work is entirely your own original work, except where clearly attributed otherwise, and that it has not been submitted partly or wholly for any other educational award.

I hereby declare that:

- this is all my own work, unless clearly indicated otherwise, with full and proper accreditation;
- with respect to my own work: none of it has been submitted at any educational institution contributing in any way to an educational award;
- with respect to another's work: all text, diagrams, code, or ideas, whether verbatim, paraphrased or otherwise modified or adapted, have been duly attributed to the source in a scholarly manner, whether from books, papers, lecture notes or any other student's work, whether published or unpublished, electronically or in print.

Suneet Mahajan
April 17, 2025

Abstract

This project explores the mathematical connection between optimal control theory and backpropagation in neural networks, and in the process, reveals their shared theoretical foundations. While backpropagation has revolutionised modern deep learning, its origins in control theory have typically remained underappreciated and overlooked. Reformulating neural network training as an optimal control problem enables this research to establish a rigorous mathematical framework that bridges these seemingly separate disciplines.

The study begins by examining the historical development of both fields, tracing their parallel evolution and conceptual similarities. Through a detailed analysis, we demonstrate that backpropagation is not merely analogous to, but mathematically equivalent to solving the adjoint equations derived from Pontryagin's Maximum Principle. This perspective allows us to reframe weight updates as control inputs that guide the network's hidden states along an optimal trajectory to minimise a cost functional.

This reformulation offers distinct, unexpected advantages beyond conventional approaches. It offers a more principled understanding of neural network training dynamics through the lens of dynamical systems theory, by providing theoretical rigour where heuristic explanations often dominate. Furthermore, it enables the application of established control-theoretic tools such as adjoint methods and Hamiltonian analysis to investigate convergence, stability, and robustness properties. Finally, it inspires novel training strategies inspired by optimal control techniques.

The project includes computational implementations that validate the theoretical framework on a supervised learning task, comparing the control-based approach with standard backpropagation. The obtained results demonstrate comparable performance, whilst providing enhanced interpretability and theoretical insights. This work aims to contribute to the growing body of research seeking deeper mathematical foundations for deep learning techniques and further emphasises the value of cross-disciplinary perspectives in the advancement of machine learning theory.

Total word count of report: 13,750

Acknowledgements

I would like to thank my supervisor, Professor Michel Schellekens, for proposing this deeply fascinating project, and for his patient support throughout the year.

I would also like to thank Alex Goodison, Daniel Kharchenko, Andrew Nash (University College Cork), Aaryan Prakash (University of Texas at Austin), Saarang Srinivasan (Purdue University) and Albert Zhu (Rice University) for taking the time to review and proofread this report.

Contents

List of Abbreviations

BPTT	Backpropagation Through Time
DP	Dynamic Programming
IVP	Initial Value Problem
LQR	Linear Quadratic Regulator
ML	Machine Learning
MPC	Model Predictive Control
NN	Neural Network
OC	Optimal Control
ODE	Ordinary Differential Equation
PMP	Pontryagin's Maximum Principle
SGD	Stochastic Gradient Descent
UDE	Universal Differential Equation

Chapter 1

Introduction

1.1 Background and Motivation

Optimal control theory is a powerful mathematical framework that deals with finding control policies that optimise a given performance criterion for dynamic systems. Originally developed in the mid 20^{th} century through advances, namely Pontryagin's Maximum Principle and Bellman's Dynamic Programming, it has found applications in aerospace, robotics, economics, and biology. More recently, a growing interest has emerged in exploring the connections between optimal control and machine learning; particularly in understanding and improving training algorithms for neural networks.

Backpropagation, the core training algorithm behind neural networks, is fundamentally a gradient-based optimisation method. Using calculus' chain rule, it computes the gradient of a loss function with respect to weights. However, the mathematical machinery underlying backpropagation also bears a striking resemblance to adjoint methods used in optimal control theory. This mathematical similarity isn't merely superficial, and raises a deeper question: can the process of training a neural network be reframed as an optimal control problem?

This project was motivated by the desire to explore this connection. By interpreting the learning dynamics of neural networks through the lens of control theory, we can potentially derive novel training algorithms, gain theoretical insights, and improve both convergence and stability. In particular, by reformulating the problem through the lens of control theory, we can potentially develop more principled training algorithms with stronger theoretical guarantees than ones provided by conventional approaches.

The broader motivation stems from the need for cross-pollination between disciplines. As machine learning models become more complex, there is increasing value in revisiting classical mathematical methods for structure and clarity. Optimal control theory offers a principled way to view neural network training as the control of a dynamical system; with the weights acting as controls, and the network state as a system evolving over time.

1.2 Problem Statement

The training of artificial neural networks is typically formulated as an optimisation problem: given a set of inputs and corresponding target outputs, with the goal being to find the set of parameters (weights and biases) that minimises a predefined loss function. The conventional gradient-based methods, particularly backpropagation, have evolved somewhat heuristically, leaving us without the kind of unified theoretical framework that might illuminate why these techniques work as well as they do.

In parallel, optimal control theory provides a rigorous mathematical framework for determining the control inputs to a dynamical system that optimise a given cost functional. The core idea is to view the evolution of the system over time as governed by differential equations, and to compute the control trajectory that leads to an optimal outcome. This project investigates the hypothesis that the training of a feedforward neural network can be interpreted as an optimal control problem, where the network's hidden states evolve over a fictitious time axis and the weights serve as control variables.

Specifically, the problem addressed in this project is to:

- Reformulate the process of neural network training as an optimal control problem;
- Derive the associated optimality conditions using Pontryagin's Maximum Principle;
- Compare these conditions with the standard backpropagation algorithm;
- Explore the potential advantages of this control-theoretic formulation, both in terms of theoretical understanding and practical performance.

This approach opens up the possibility of applying established techniques from control theory, such as adjoint state methods and forward-backward differential equations, to neural network training. By bridging the conceptual gap between these two domains, this work aims to contribute to the growing body of research that seeks deeper theoretical insights into the learning process of neural networks.

1.3 Objectives of the Project

The primary aim of this project is to explore the intersection between optimal control theory and the training of artificial neural networks, with a particular focus on interpreting backpropagation as an instance of a more general control-theoretic framework.

To achieve this, the project was guided by the following objectives:

- **Reformulate neural network training as an optimal control problem:** Construct a mathematical model in which the evolution of a neural network's hidden states is governed by dynamical equations, with weights treated as control variables.
- **Apply Pontryagin's Maximum Principle to derive optimality conditions:** Use continuous-time optimal control theory to derive necessary conditions for minimising the training loss, and investigate the structure and behaviour of the resulting equations.
- **Compare the derived optimality conditions with backpropagation:** Analyse the similarities and differences between the control-theoretic approach and standard backpropagation, highlighting any new insights that emerge.
- **Implement and test the proposed formulation numerically:** Use a forward-backward sweep method to solve the control problem and apply it to a simple supervised learning task to validate the approach.
- **Reflect on theoretical implications and practical relevance:** Assess whether this reformulation offers any practical benefits in terms of training stability, convergence behaviour, or interpretability, and outline possible future directions.

These objectives collectively aim to strengthen the conceptual bridge between classical optimal control theory and modern deep learning practice, contributing both to theoretical understanding and potential algorithmic development.

1.4 Scope and Limitations

This project focuses on establishing a theoretical and computational framework for viewing neural network training through the lens of optimal control theory. In particular, it considers a simplified setting where the network is treated as a dynamical system evolving over continuous time, and the training process is framed as the task of determining an optimal control trajectory (i.e., weight configuration) that minimises a prescribed cost functional.

Scope

- The project is confined to feedforward neural networks with fixed architecture and deterministic dynamics.
- A continuous-time formulation is adopted for the theoretical derivations, though the implementation uses a discretised version for numerical solution.
- The objective function is assumed to be differentiable and standard (e.g., mean squared error or cross-entropy), with no explicit regularisation.
- The project includes an experimental demonstration on a simple supervised learning task to validate the theoretical model and numerical method.
- The analysis is primarily theoretical, with implementation designed to illustrate feasibility, rather than to compete with state-of-the-art performance.

Limitations

- The approach does not generalise immediately to recurrent or convolutional neural networks, though the underlying principles may still apply.
- The use of continuous-time models introduces certain abstractions that may not directly reflect the discrete-time nature of practical training algorithms.
- Computational efficiency is not optimised in this implementation; the focus is instead on correctness and clarity of the method.
- The experimental results are limited in scope and are not intended as a benchmark against conventional training approaches.
- No formal stability or convergence guarantees are proven for the numerical solution method employed.

These boundaries were necessary to maintain focus and feasibility within the scope of an undergraduate research project, but they also leave open several avenues for future work and refinement.

1.5 Report Structure

The remainder of this report is structured as follows:

- **Chapter 2 – Literature Review:** This chapter surveys existing work on optimal control theory and its connections to machine learning. It reviews the development of backpropagation, adjoint methods, and recent efforts to unify these ideas under a control-theoretic framework.

- **Chapter 3 – Mathematical Preliminaries:** This chapter introduces the mathematical foundations necessary for the theoretical development, including continuous-time dynamical systems, Pontryagin's Maximum Principle, and adjoint sensitivity analysis.
- **Chapter 4 – Training Neural Networks as an Optimal Control Problem:** The supervised learning task is reformulated as an optimal control problem. A feedforward neural network is treated as a dynamical system, and training is cast as the problem of optimally controlling this system to minimise a cost functional.
- **Chapter 5 – Optimal Control Methods in Deep Learning:** This chapter derives the optimality conditions using tools from control theory and presents the numerical approach based on the forward-backward sweep method. A discretised version of the continuous-time formulation is used for implementation.
- **Chapter 6 – Experiments and Numerical Examples:** This chapter presents two computational case studies. First, the control-theoretic training method is applied to a supervised learning task and compared with standard backpropagation. Second, a neural network is used to learn system dynamics, which are then deployed in a model predictive control (MPC) framework to evaluate control performance.
- **Chapter 7 – Discussion and Conclusions:** The final chapter synthesises theoretical and empirical findings, discusses implications and limitations, and suggests promising directions for future research.

Chapter 2

Literature Review

2.1 Historical Background and Prior Work

The development of neural networks and optimal control theory has largely occurred in parallel, with limited interaction until recent years. Nevertheless, both fields are deeply rooted in optimisation and the study of dynamical systems.

The foundations of *artificial neural networks* trace back to the 1940s with the work of McCulloch and Pitts, and later the introduction of the perceptron by Rosenblatt in 1958. However, it was not until the 1980s that neural networks gained widespread traction, thanks in part to the rediscovery of *backpropagation* as a practical training algorithm for multilayer perceptrons. Backpropagation, which computes gradients of the loss function efficiently via the chain rule, became the cornerstone of modern deep learning.

In parallel, *optimal control theory* was being developed to address problems in engineering and physics where the goal is to determine control inputs that steer a dynamical system in an optimal way. The seminal work of Pontryagin et al. (1962) in the late 1950s introduced Pontryagin's Maximum Principle, a key result for solving continuous-time control problems using *adjoint (co-state) variables* and necessary conditions for optimality. Around the same time, foundational contributions such as Kelley (1960)'s gradient theory of optimal flight paths and Halkin (1964, 1966)'s results on discrete-time systems helped to generalise optimality principles to both continuous and discrete settings.

Despite the similarities between the use of adjoints in control theory and gradients in machine learning, the two fields remained largely separate. It wasn't until more recently, with the rise of *deep learning theory*, that researchers began to draw formal parallels. In particular, the interpretation of backpropagation as a *discretised adjoint method* has sparked renewed interest in applying control-theoretic tools to neural network training.

Some notable prior works include:

- LeCun (1988), who analysed backpropagation in terms of gradient flows;
- Werbos (1990), who explicitly linked backpropagation with *dynamic programming* and control;
- Recent advances in *neural ODEs* Chen et al. (2018), where the forward pass of a network is modelled as a continuous-time dynamical system and gradients are computed using the *adjoint sensitivity method*, a direct application of control principles.

These developments provide the theoretical basis and motivation for this project's exploration of neural network training as an optimal control problem.

2.2 Optimal Control Theory in Machine Learning

The intersection of optimal control theory and machine learning has become increasingly prominent in recent years, particularly as researchers seek principled ways to understand and improve the training dynamics of deep neural networks.

At a high level, both fields involve the optimisation of some objective under constraints imposed by a dynamical system. In the case of machine learning, the system dynamics correspond to how inputs propagate through layers of a neural network, and the objective is to minimise a loss function defined over prediction errors. In optimal control, the dynamics are governed by differential equations, and the objective is to determine control inputs that guide the system towards a desired state while minimising a cost.

This structural similarity has led to a number of recent efforts to explicitly formulate machine learning tasks as control problems:

- In *neural ordinary differential equations (neural ODEs)*, the forward pass of a neural network is described as a continuous-time dynamical system $\dot{x}(t) = f(x(t), u(t))$, with parameters $u(t)$ acting as control variables. Training such models involves solving an optimal control problem, where the loss depends on the terminal state.
- *Optimal control theory* provides a natural framework for understanding backpropagation in continuous time via *adjoint sensitivity analysis*, which is mathematically equivalent to the costate dynamics in Pontryagin's Maximum Principle.
- *Gradient-based methods* widely used in machine learning can be seen as approximations of the continuous-time optimality conditions derived from control theory. This offers a new lens through which to analyse convergence, stability, and robustness.
- In some formulations, *control inputs are interpreted as weight trajectories* over time, and regularisation can be viewed as penalising control effort. This allows new forms of regularisation to be introduced via the control cost functional.

These insights not only enrich the theoretical understanding of learning algorithms but also offer potential for new training strategies inspired by classical control techniques, including bang-bang control, feedback stabilisation, and indirect optimisation methods.

This project builds on this perspective by formulating the training of a simple neural network as an optimal control problem, explicitly deriving the optimality conditions using Pontryagin's Maximum Principle, and comparing them with standard backpropagation updates.

2.3 Comparison with Classical Learning Approaches

Conventional training of neural networks relies on gradient-based optimisation algorithms, such as stochastic gradient descent (SGD) and its variants. These methods operate in discrete time, iteratively updating network parameters based on the gradient of a loss function computed over a dataset. This paradigm, while highly effective, treats the learning process as a static optimisation problem over a high-dimensional parameter space.

In contrast, the *optimal control perspective* reframes the problem as a dynamic optimisation task, where network weights evolve continuously as control inputs that shape the trajectory of an underlying dynamical system. Rather than directly optimising the parameters at each layer, the aim is to find a time-varying control signal that minimises a global cost functional subject to system dynamics.

Some key differences between the two paradigms are outlined below:

Feature	**Classical Learning**	**Optimal Control Formulation**
Nature of Optimisation	Static optimisation	Dynamic optimisation with state/control variables
Gradient Computation	Backpropagation (chain rule)	Adjoint method (Pontryagin's principle)
Time Discretisation	Implicit (layered architecture)	Explicit continuous/discrete time system
Regularisation	Explicit (e.g., L2 penalty)	Integrated into cost functional as control cost
Perspective on Weights	Fixed parameters	Time-dependent control inputs

Table 2.1: Comparison between classical learning and optimal control perspectives

Despite these differences, both approaches are mathematically linked. The backpropagation algorithm, when applied to neural ODEs, is equivalent to solving the adjoint equations of the continuous-time optimal control problem. In this sense, *classical learning algorithms can be interpreted as special cases* of broader control-based formulations.

This connection opens up a new set of tools for analysing and improving training algorithms. For instance, the optimal control framework enables the incorporation of constraints, multi-objective costs, and feedback mechanisms; features that are less naturally expressed in traditional training setups.

Understanding these relationships is not only of theoretical interest but also offers practical value. For example, control-based training may provide advantages in stability, robustness, and interpretability, particularly for systems that are naturally continuous in time or subject to external constraints.

Chapter 3

Mathematical Preliminaries

3.1 Mathematical Formulation of Neural Network Training

At its core, the training of a neural network is an optimisation problem; given a set of input-output pairs (x_i, y_i), the goal is to find parameters (weights and biases) θ that minimise the discrepancy between the network's output $f_\theta(x_i)$ and the true label y_i, according to a loss function L.

For a dataset $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^N$, the standard formulation is:

$$\min_\theta \ \mathcal{J}(\theta) = \frac{1}{N} \sum_{i=1}^{N} L(f_\theta(x_i), y_i)$$

This is a high-dimensional, non-convex optimisation problem. In classical approaches, this is typically solved using gradient descent:

$$\theta_{k+1} = \theta_k - \eta \nabla_\theta \mathcal{J}(\theta_k)$$

where η is the learning rate.

To recast this in an optimal control framework, we consider the neural network as a discrete or continuous-time *dynamical system*. The forward propagation of activations through the layers corresponds to the system dynamics. For a continuous-time model such as a neural ODE, this takes the form:

$$\frac{d}{dt} x(t) = f(x(t), u(t)), \quad x(0) = x_0$$

Here:

- $x(t)$ represents the state (activations),
- $u(t)$ represents the control input (parameters),
- f is the system dynamics defined by the neural network's architecture.

The training objective is then to find a control input $u(t)$ that drives the system to a desired terminal state $x(T)$, minimising a cost functional of the form:

$$\mathcal{J}(u) = \Phi(x(T)) + \int_0^T \mathcal{L}(x(t), u(t))\, dt$$

where Φ measures terminal loss (e.g. prediction error) and $\mathcal{L}$ includes potential regularisation terms (e.g. control effort).

This perspective aligns training with *trajectory optimisation* in control theory, where weights are seen as time-dependent variables, and loss is accumulated over a trajectory.

Such a formulation allows for the application of analytical tools from control theory, including Pontryagin's Maximum Principle and Hamiltonian analysis, which will be explored in the next section.

3.2 Pontryagin's Maximum Principle

Pontryagin's Maximum Principle (PMP) is a cornerstone of optimal control theory. It provides necessary conditions for a control function to be optimal in problems where the objective is to minimise a cost functional subject to dynamical constraints.

Let us consider a general control system of the form:

$$\frac{d}{dt}x(t) = f(x(t), u(t)), \quad x(0) = x_0$$

with cost functional:

$$\mathcal{J}(u) = \Phi(x(T)) + \int_0^T \mathcal{L}(x(t), u(t))\, dt$$

PMP introduces an auxiliary variable, the *adjoint state* (or co-state) $\lambda(t)$, and defines the *Hamiltonian*:

$$\mathcal{H}(x, u, \lambda) = \lambda^\top f(x, u) + \mathcal{L}(x, u)$$

According to PMP, if $u^*(t)$ is an optimal control and $x^*(t)$ is the corresponding trajectory, then there exists an adjoint state $\lambda(t)$ satisfying:

$$\frac{d}{dt}\lambda(t) = -\frac{\partial \mathcal{H}}{\partial x} \quad \text{with terminal condition} \quad \lambda(T) = \frac{\partial \Phi}{\partial x}(x(T))$$

Furthermore, the optimal control must maximise the Hamiltonian pointwise:

$$u^*(t) = \arg\max_u \mathcal{H}(x^*(t), u, \lambda(t))$$

In the context of neural networks:

- $x(t)$ represents the activations as they propagate through a continuous network (e.g. a neural ODE),
- $u(t)$ represents the weights (control inputs),
- $\lambda(t)$ corresponds to backpropagated gradients (adjoint variables),
- the Hamiltonian encodes both the system dynamics and the learning objective.

And thus, PMP provides a principled way to derive the training dynamics of a network, offering a continuous-time analogue to the backpropagation algorithm. In fact, for neural ODEs, the standard adjoint sensitivity method used in practice emerges as a direct consequence of applying PMP.

This framework is especially powerful when dealing with constrained or structured learning problems, for example, enforcing bounded controls, hard constraints on states, or time-dependent objectives.

3.3 Hamiltonian Dynamics in Control Problems

In optimal control theory, the *Hamiltonian function* plays a central role in connecting the system dynamics, control inputs, and the cost functional into a single framework. As introduced in the previous section, the Hamiltonian for a control system is defined as:

$$\mathcal{H}(x, u, \lambda) = \lambda^\top f(x, u) + \mathcal{L}(x, u)$$

This function serves as the analogue of the classical Hamiltonian in physics, with the adjoint state $\lambda(t)$ resembling a generalised momentum.

The *Hamiltonian system* arising from Pontryagin's Maximum Principle consists of a set of first-order differential equations describing both the forward evolution of the state and the backward evolution of the adjoint variables:

$$\frac{d}{dt}x(t) = \frac{\partial \mathcal{H}}{\partial \lambda}, \qquad \frac{d}{dt}\lambda(t) = -\frac{\partial \mathcal{H}}{\partial x}$$

These are sometimes referred to as the *canonical equations* of optimal control.

3.3.1 Interpretation in Learning Problems

In the context of neural networks and machine learning:

- The state $x(t)$ represents the intermediate computations (activations),
- The adjoint state $\lambda(t)$ represents error signals or gradients that flow backward through the system,
- The Hamiltonian governs how these interact during learning.

This dual system mirrors the forward and backward passes of traditional backpropagation:

- The forward pass computes $x(t)$ via the dynamics,
- The backward pass computes $\lambda(t)$ via adjoint dynamics.

By explicitly formulating learning as a Hamiltonian system, we gain theoretical tools for analysing stability, energy conservation, and sensitivity to initial conditions — insights that are crucial when considering training efficiency, generalisation, and robustness.

3.3.2 Relevance to Neural ODEs and Beyond

In continuous-depth models like neural ODEs, Hamiltonian formulations can guide the design of reversible architectures, energy-conserving dynamics, and symplectic integrators. Some models, such as *Hamiltonian Neural Networks*, are explicitly designed to respect these principles.

In constrained learning problems (e.g. trajectory tracking, energy-constrained control), the Hamiltonian approach also allows for explicit encoding of constraints through methods like augmented Lagrangians.

Altogether, the Hamiltonian formalism gives a powerful geometric and physical interpretation of learning, laying the groundwork for deeper control-theoretic approaches to machine learning.

Chapter 4

Training Neural Networks as an Optimal Control Problem

4.1 Training Neural Networks as an Optimal Control Problem

4.1.1 Neural Networks as Discrete-Time Dynamical Systems

A feedforward neural network can be naturally viewed as a *discrete-time dynamical system*, where each layer represents a time step in the evolution of an input through a sequence of transformations.

Consider a network with L layers, where the state at layer l is denoted by x_l, and the control variables are the weights u_l. The forward dynamics can be expressed recursively as:

$$x_{l+1} = \sigma(W_l x_l + b_l) = f_l(x_l, u_l), \quad l = 0, \ldots, L-1$$

Here:

- x_0 is the input to the network,
- $\sigma(\cdot)$ is a nonlinear activation function,
- $u_l = (W_l, b_l)$ are the control parameters for layer l.

This recurrence defines a discrete trajectory of states $\{x_1, x_2, \ldots, x_L\}$, mapping the input to an output. The goal of training is to find optimal parameters $\{u_l\}$ that minimise a cost function, typically a loss $\mathcal{L}(x_L, y)$, where y is the target.

4.1.2 Cost Functional

The training objective can be expressed as a sum over layer-wise costs (including regularisation):

$$\mathcal{J}(u) = \mathcal{L}(x_L, y) + \sum_{l=0}^{L-1} R(u_l)$$

where $R(u_l)$ is a regularisation term (e.g. weight decay). This resembles the finite-horizon cost structure found in optimal control problems Bertsekas (2017).

4.1.3 Backpropagation as an Adjoint Method

The backward pass of training, namely *backpropagation*, corresponds to computing the adjoint variables $\lambda_l = \frac{\partial \mathcal{J}}{\partial x_l}$ using the chain rule. This backward recursion mirrors the discrete-time adjoint equations in optimal control theory:

$$\lambda_l = \left(\frac{\partial f_l}{\partial x_l}\right)^\top \lambda_{l+1} + \frac{\partial R}{\partial x_l}$$

This interpretation reveals that the standard training algorithm for neural networks is a special case of a more general framework from control theory, where we optimise a control sequence (weights) to guide the system (input states) to a desired output.

4.2 Neural ODEs and Continuous-Time Models

While traditional neural networks operate as discrete-time dynamical systems, an emerging class of models — *Neural Ordinary Differential Equations (Neural ODEs)* — reinterpret the forward pass of a neural network as a *continuous-time dynamical system*.

Instead of computing layer-wise updates of the form:

$$x_{l+1} = f_l(x_l, u_l),$$

a Neural ODE models the evolution of the hidden state $x(t)$ as the solution to an ordinary differential equation:

$$\frac{dx(t)}{dt} = f(x(t), u(t)), \quad x(0) = x_0.$$

Here, the function f (parameterised by a neural network) defines the dynamics of the system over time. The final output of the network corresponds to $x(T)$, the solution of the ODE at time T, typically computed using numerical ODE solvers.

4.2.1 Motivation and Advantages

This paradigm shift introduces several conceptual and practical advantages:

- **Parameter efficiency:** Unlike deep residual networks with many layers, Neural ODEs allow a continuous-depth model with shared parameters across time.
- **Memory efficiency:** Using reversible dynamics and adjoint sensitivity methods, gradients can be computed with low memory overhead.
- **Adaptive computation:** ODE solvers adapt the number of function evaluations based on desired accuracy, allowing for input-dependent depth.

Furthermore, in the context of optimal control, this formulation aligns naturally with the *continuous-time control problem*, allowing for the use of powerful tools from control theory.

4.2.2 Training via the Adjoint Method

To compute gradients for training, the adjoint sensitivity method is employed. For a loss function $\mathcal{L}(x(T))$, the gradient with respect to the parameters u is computed by solving a second, backward-in-time ODE:

$$\frac{d\lambda(t)}{dt} = -\lambda(t)^\top \frac{\partial f}{\partial x}, \quad \lambda(T) = \frac{\partial \mathcal{L}}{\partial x(T)}.$$

This is equivalent to solving the *Hamiltonian adjoint system*, where the adjoint state $\lambda(t)$ plays the role of the co-state in Pontryagin's framework.

Thus, training a Neural ODE becomes a classic *continuous-time optimal control problem*: optimise the control input $u(t)$ to drive the system from initial state $x(0)$ to a target state $x(T)$, while minimising a cost.

4.3 Backpropagation as a Special Case of Optimal Control

One of the key insights of framing deep learning as an optimal control problem is that the widely used *backpropagation algorithm* emerges as a specific instance of a broader class of *adjoint-based optimisation techniques* used in control theory.

4.3.1 Backpropagation and Adjoint Equations

In optimal control, given a dynamical system with states $x(t)$, controls $u(t)$, and a terminal cost $\mathcal{L}(x(T))$, the necessary conditions for optimality are derived using *Pontryagin's Maximum Principle*. The principle introduces *co-state (adjoint) variables* $\lambda(t)$, governed by backward differential equations:

$$\frac{d\lambda(t)}{dt} = -\left(\frac{\partial f}{\partial x}\right)^\top \lambda(t), \quad \lambda(T) = \frac{\partial \mathcal{L}}{\partial x(T)}.$$

This is mathematically equivalent to the *reverse-mode differentiation* carried out in backpropagation, where:

- $x(t)$ represents the forward-pass activations,
- $\lambda(t)$ are the gradients of the loss with respect to intermediate activations,
- The backward differential equation governs how the gradient signal is propagated.

Thus, the backpropagation algorithm is an efficient implementation of the *adjoint method* in discrete time.

4.3.2 Implications of the Control Perspective

This reinterpretation allows for deeper theoretical understanding and opens up the possibility of alternative training methods. For example:

- **Continuous-time training** can be viewed as solving a boundary value problem (BVP), which may allow for better long-term gradient stability.
- **Control constraints** can be explicitly incorporated, such as bounds on weights or sparsity constraints.
- **Multi-objective training** can be handled more naturally by defining multi-term cost functionals.

Furthermore, this formalism provides a natural pathway for adapting traditional optimal control techniques — such as *shooting methods*, *direct collocation*, and *indirect optimisation* — to neural network training.

Chapter 5

Optimal Control Methods in Deep Learning

5.1 The Direct Method: Discretise-Then-Optimise

In the context of optimal control, the *direct method* refers to an approach where the continuous-time control problem is first discretised, transforming it into a finite-dimensional nonlinear programming (NLP) problem Biegler (2010). The resulting discrete optimisation problem is then solved using standard optimisation techniques such as gradient descent, L-BFGS, or other numerical solvers.

This approach aligns closely with how most machine learning algorithms operate in practice — by *sampling trajectories*, *unrolling models in time*, and *optimising discretised objective functions* using gradient-based methods Li et al. (2017); Degrave et al. (2018).

5.1.1 Discretisation of the Dynamics

Given the continuous-time dynamics:

$$\frac{dx(t)}{dt} = f(x(t), u(t)), \quad x(0) = x_0,$$

the time interval $[0, T]$ is divided into N uniform steps $\{t_0, t_1, \ldots, t_N\}$, with step size $\Delta t = \frac{T}{N}$. Using the *Euler method* or other integration schemes, the system can be approximated as:

$$x_{k+1} = x_k + \Delta t \cdot f(x_k, u_k),$$

with x_0 given.

The objective functional is similarly discretised:

$$\mathcal{J}(u) = \sum_{k=0}^{N-1} \ell(x_k, u_k) + \phi(x_N),$$

where ℓ is the running cost and ϕ is the terminal cost.

5.1.2 Advantages and Use in Deep Learning

The direct method is particularly well suited to training neural networks, where the forward pass naturally corresponds to a discretised dynamical system (especially in models like ResNets and RNNs) Haber and Ruthotto (2017). Its key advantages include:

- **Straightforward implementation** using automatic differentiation and off-the-shelf optimisers.
- **Scalability** to high-dimensional parameter spaces.
- **Compatibility with stochastic optimisation** techniques such as SGD and Adam.

Additionally, the direct method provides a clear interpretation of *gradient descent as an optimal control solution* to a discretised trajectory optimisation problem.

5.2 The Indirect Method: Optimise-Then-Discretise

In contrast to the direct method, the *indirect method* involves first deriving the *necessary optimality conditions* for the continuous-time control problem analytically, before discretising the resulting system of equations for numerical solution. These conditions are typically derived using *Pontryagin's Maximum Principle (PMP)* Pontryagin et al. (1962); Kirk (2004) or the *calculus of variations*.

This approach focuses on the fundamental structure of the control problem and often leads to a deeper understanding of the solution's dynamics and sensitivity.

5.2.1 Formulation Using Pontryagin's Maximum Principle

Given the system:

$$\frac{dx(t)}{dt} = f(x(t), u(t)), \quad x(0) = x_0,$$

and the objective:

$$\mathcal{J}(u) = \int_0^T \ell(x(t), u(t))\, dt + \phi(x(T)),$$

we introduce the *Hamiltonian*:

$$\mathcal{H}(x, u, \lambda) = \ell(x, u) + \lambda^\top f(x, u),$$

where $\lambda(t)$ are the adjoint (co-state) variables. The necessary conditions for optimality are:

- **State dynamics:** $\frac{dx(t)}{dt} = \frac{\partial \mathcal{H}}{\partial \lambda}$
- **Adjoint dynamics:** $\frac{d\lambda(t)}{dt} = -\frac{\partial \mathcal{H}}{\partial x}$
- **Optimal control:** $u^*(t) = \arg\min_u \mathcal{H}(x, u, \lambda)$

This results in a *two-point boundary value problem (BVP)*: the state evolves forward in time with initial conditions, while the co-state evolves backward in time from terminal conditions Bryson and Ho (1975).

5.2.2 Challenges and Relevance to Machine Learning

While powerful, the indirect method is often more difficult to implement in practice due to:

- The need to solve boundary value problems numerically, which can be unstable.
- Sensitivity to initial guesses and convergence issues.

- Difficulty scaling to high-dimensional systems like deep networks.

Nonetheless, the *conceptual insights* offered by the indirect method have been influential. For example, the interpretation of *backpropagation as solving an adjoint equation* is directly inspired by this approach.

Moreover, advances in *neural differential equations* and *physics-informed machine learning* have renewed interest in applying indirect optimal control methods, especially when physical consistency and interpretability are important.

5.3 Shooting Methods

Shooting methods are a class of techniques used to solve optimal control problems by converting them into boundary value problems (BVPs). These methods are particularly useful in the context of the *indirect approach*, where optimality conditions (e.g., from Pontryagin's Maximum Principle) yield differential equations for the state and adjoint variables.

The central idea is to "shoot" from the initial condition and iteratively adjust the control or co-state variables to satisfy the terminal conditions, similar to aiming a projectile to hit a target.

5.3.1 Single Shooting

In single shooting, the control inputs (or co-state variables) are parameterised, and the system is forward integrated from the initial state. An optimisation routine adjusts the parameters to minimise the deviation from the terminal condition.

- **Advantages:** Conceptually simple; easy to implement.
- **Disadvantages:** Highly sensitive to initial guesses; can be numerically unstable, especially for stiff or long-horizon problems.

5.3.2 Multiple Shooting

To improve stability, multiple shooting breaks the time horizon into segments. Each segment is treated as an initial value problem with its own initial state variable. Continuity constraints are introduced to ensure that the segments stitch together smoothly Bock and Plitt (1984).

- **Advantages:** Better numerical stability and convergence properties.
- **Disadvantages:** Involves more variables and constraints, increasing the problem size.

5.3.3 Relevance to Deep Learning

Shooting methods have strong analogies with training recurrent or unrolled models in machine learning:

- **Single shooting** is similar to backpropagation through time (BPTT) using one long forward pass.
- **Multiple shooting** is conceptually closer to truncated BPTT, where shorter segments are optimised individually, potentially improving gradient flow and stability.

These connections suggest that techniques from optimal control, such as multiple shooting, can inform the design of more stable training procedures for deep sequential or continuous-time models.

5.4 The Adjoint Method and Backpropagation

The adjoint method is a cornerstone of optimal control theory and underpins many modern algorithms in deep learning, particularly *backpropagation*. It allows for efficient computation of gradients in systems governed by differential equations, and it is essential for training neural ODEs and control-based learning models.

5.4.1 The General Principle

Given a loss functional of the form:

$$\mathcal{J}(u) = \int_0^T \ell(x(t), u(t))\, dt + \phi(x(T)),$$

subject to the dynamics:

$$\frac{dx(t)}{dt} = f(x(t), u(t)), \quad x(0) = x_0,$$

the adjoint method introduces a Lagrange multiplier $\lambda(t)$ to enforce the dynamics as a constraint Pontryagin et al. (1962). The Lagrangian becomes:

$$\mathcal{L} = \mathcal{J}(u) + \int_0^T \lambda(t)^\top \left(\frac{dx(t)}{dt} - f(x(t), u(t)) \right) dt.$$

Taking variations and integrating by parts yields the adjoint equation:

$$\frac{d\lambda(t)}{dt} = -\frac{\partial \mathcal{H}}{\partial x}, \quad \lambda(T) = \frac{\partial \phi}{\partial x}(x(T)),$$

and the gradient of the loss with respect to controls:

$$\frac{d\mathcal{J}}{du} = \frac{\partial \mathcal{H}}{\partial u},$$

where $\mathcal{H}$ is the Hamiltonian.

5.4.2 Connection to Backpropagation

Backpropagation is an application of the adjoint method to discrete-time layered systems. In a feedforward neural network, the state equations correspond to layer-wise computations:

$$x_{k+1} = f_k(x_k, \theta_k),$$

and the adjoint variables propagate gradients backward from the loss through the network:

$$\lambda_k = \frac{\partial \ell}{\partial x_k} + \frac{\partial f_k}{\partial x_k}^\top \lambda_{k+1}.$$

In this sense, backpropagation can be viewed as a discrete adjoint method, with each layer representing a time step. This analogy has been formalised in continuous-time settings through neural ODEs, where gradients are computed using the continuous adjoint method.

5.4.3 Benefits and Limitations

Benefits:

- Scales well with large parameter spaces.
- Enables efficient gradient-based optimisation in both classical control and modern machine learning.

Limitations:

- In continuous-time models, the adjoint method can suffer from numerical instability or memory issues.
- Requires differentiability and may not handle discontinuities or stiff systems well.

5.5 Neural ODEs as Dynamical Systems

Neural Ordinary Differential Equations (Neural ODEs) provide a continuous-time analogue of deep neural networks by modelling the evolution of hidden states using differential equations parameterised by neural networks. Introduced by Chen et al. (2018), they formalise the forward pass as solving an initial value problem (IVP).

The Core Idea

Instead of a finite sequence of layers as in residual networks (ResNets), Neural ODEs define:

$$\frac{dx(t)}{dt} = f(x(t), t, \theta), \quad x(0) = x_0,$$

where f is a neural network with parameters θ. The output at time T is obtained by integrating the dynamics:

$$x(T) = x_0 + \int_0^T f(x(t), t, \theta)\, dt.$$

This framework generalises residual connections:

$$x_{k+1} = x_k + f_k(x_k) \quad \text{(ResNet)} \quad \rightarrow \quad \frac{dx}{dt} = f(x, t, \theta) \quad \text{(Neural ODE)}.$$

5.5.1 Advantages

- **Adaptive computation:** ODE solvers can choose time steps adaptively, potentially improving efficiency.
- **Parameter efficiency:** Replaces depth with continuous dynamics, reducing the number of layers needed.
- **Interpretable latent trajectories:** Useful in applications like time series and physical systems.

5.5.2 Training Neural ODEs

Training involves computing gradients through the ODE solver. This can be done using:

- **Adjoint sensitivity method:** Integrates backward in time to compute gradients efficiently.
- **Backpropagation through the solver:** Differentiates through all solver steps (more accurate but memory-intensive).

5.5.3 Connection to Optimal Control

Neural ODEs interpret learning as a continuous-time control problem:

$$\min_{\theta} \mathcal{J} = \mathcal{L}(x(T)), \quad \text{subject to} \quad \frac{dx}{dt} = f(x, t, \theta),\ x(0) = x_0.$$

This directly mirrors an optimal control problem with: θ acting as the control input, f defining the system dynamics, and $\mathcal{L}$ representing the terminal cost.

Thus, training Neural ODEs can be seen as solving a continuous-time optimal control problem; an interpretation that motivates the use of adjoint methods and control-theoretic tools in machine learning.

5.6 Continuous-Time Optimal Control Formulations of Learning Problems

Many modern learning problems, especially those involving deep learning models like neural ODEs, can be formulated naturally within the framework of continuous-time optimal control. In these formulations, learning is interpreted as the optimisation of a control policy over time, where the dynamics of the system are defined by neural networks.

5.6.1 General Setup

Consider a supervised learning problem where a model is trained to map inputs to outputs. In the continuous-time control framework, the model's dynamics are given by:

$$\frac{dx(t)}{dt} = f(x(t), u(t)), \quad x(0) = x_0,$$

where:

- $x(t)$ is the state of the system (e.g., a hidden representation),
- $u(t)$ is the control input (e.g., parameters of a neural network),
- f is a neural network representing the system's evolution.

The learning objective can be expressed as a cost functional:

$$\mathcal{J}(u) = \int_0^T \ell(x(t), u(t))\, dt + \phi(x(T)),$$

where:

- ℓ is an instantaneous loss (e.g., cross-entropy),
- ϕ is a terminal cost measuring prediction error at the output.

5.6.2 Benefits of This Perspective

This viewpoint brings a variety of advantages:

- **Unified framework:** Many deep learning problems can be viewed as control problems with learnable dynamics and control laws.
- **Rich mathematical structure:** Enables use of Hamiltonians, adjoint equations, and control constraints in the training process.
- **Analytical tools:** Allows theoretical insights into stability, convergence, and regularisation.

5.6.3 Applications in Deep Learning

Several recent developments have applied this framework to practical models:

- **Neural ODEs:** As previously discussed, these models directly follow the continuous-time formulation.
- **Meta-learning:** The adaptation of parameters over time can be seen as a control process.
- **Deep reinforcement learning:** Policy and value functions evolve under dynamics that can be framed in optimal control terms.

5.6.4 Challenges

Despite its elegance, this approach also introduces challenges:

- **Computational cost:** Solving differential equations and adjoint systems can be expensive.
- **Numerical stability:** Continuous-time models may exhibit instability or be sensitive to solver accuracy.
- **Interpretability:** Control-theoretic parameters do not always have intuitive interpretations in machine learning contexts.

5.7 Optimal Control as Regularisation

One powerful perspective in machine learning views regularisation not just as a heuristic for improving generalisation, but as a natural outcome of optimal control theory. In this view, regularisation terms correspond to penalties in the optimal control problem that guide the system to behave in a desired way.

5.7.1 Regularisation in Learning

In traditional supervised learning, we often add regularisation terms to the loss function:

$$\mathcal{J}(\theta) = \mathcal{L}(\theta) + \lambda \mathcal{R}(\theta),$$

where:

- $\mathcal{L}$ is the original loss (e.g., classification error),
- $\mathcal{R}$ is a regularisation term (e.g., L_2 norm),
- λ controls the strength of the penalty.

This can be reinterpreted in the optimal control framework as a cost functional that penalises undesirable trajectories or control strategies.

5.7.2 Control-Based Interpretation

Given a dynamical system:

$$\frac{dx(t)}{dt} = f(x(t), u(t)), \quad x(0) = x_0,$$

we define the cost functional as:

$$\mathcal{J}(u) = \int_0^T \ell(x(t), u(t))\, dt + \phi(x(T)),$$

where regularisation enters through $\ell(x, u)$. For example:

$$\ell(x, u) = \mathcal{L}(x) + \lambda \|u(t)\|^2$$

penalises large control inputs, encouraging smoother or more physically plausible trajectories Chizat et al. (2017). In machine learning terms, this discourages large weights or rapid changes in activations.

Examples of Regularisation via Control

- **Weight decay** corresponds to an L_2 control penalty: $\|u(t)\|^2$.
- **Path smoothness** can be encouraged by penalising $\|\dot{x}(t)\|^2$, leading to smoother hidden state dynamics in models like Neural ODEs.
- **Sparsity** can be introduced with L_1 penalties on $u(t)$.

5.7.3 Insights and Benefits

- Regularisation terms gain a principled interpretation: they are not merely add-ons, but reflect preferences over the system's evolution.
- They help ensure well-posedness in continuous-time models, preventing trajectories from diverging.
- This opens the door to more advanced control-theoretic penalties, including state constraints, energy budgets, or safety requirements.

By embedding regularisation directly into the control cost, we align the learning objective with desired dynamical behaviour, making training more robust and interpretable.

Chapter 6

Experiments and Numerical Examples

6.1 Resource Allocator

Before exploring the main experiments involving neural networks and learned dynamics, we begin with a simple illustrative example that highlights the use of optimal control principles in a classical setting. The goal is to allocate a fixed quantity of resources between three competing processes, each with a different non-linear reward function. The problem is solved using a constrained optimisation approach based on the structure of Pontryagin's Maximum Principle.

The output of each process is modelled as:

- $\sqrt{x_1}$ for Process 1 (diminishing returns),
- $\log(1 + x_2)$ for Process 2 (logarithmic gain),
- $\sqrt[3]{x_3}$ for Process 3 (sub-linear response).

subject to the constraint $x_1 + x_2 + x_3 = R$, where R is the total available resource.

The optimisation is implemented in Python using `scipy.optimize.minimize`, with constraints and non-linear reward functions encoded directly:

```
def objective(x):
    return -(np.sqrt(x[0]) + np.log(1 + x[1]) + np.cbrt(x[2]))

def resource_constraint(x):
    return np.sum(x) - R  # Total resources must sum to R

initial_guess = [R/3, R/3, R/3]
bounds = [(0, R), (0, R), (0, R)]
constraints = {'type': 'eq', 'fun': resource_constraint}

result = minimize(objective, initial_guess, bounds=bounds, constraints=
    constraints)
```

Listing 6.1: Core logic for optimal resource allocation.

This code (an excerpt from `resource_allocator.py`) reflects a classical constrained optimisation problem, where the aim is to distribute resources across three non-linear systems

in a way that maximises total benefit. The control input here is the vector of resource allocations, and the constraint ensures total resource conservation. A visualisation of the resulting output surface, as a function of allocations to Processes 1 and 2, is shown in Figure 6.1.

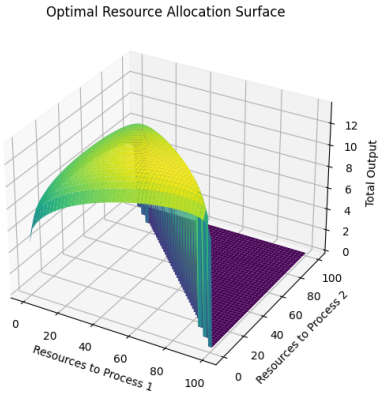

Figure 6.1: Visualisation of total output as a function of resource allocations to Processes 1 and 2, assuming a fixed total budget $R = 100$.

While simple, this example demonstrates the kinds of non-linear optimisation problems that arise in control theory, and how constraints on inputs (i.e., control variables) can be handled analytically or numerically. We now transition to experiments focused on neural network training as an optimal control problem.

6.2 Numerical Example: Linear Quadratic Regulator (LQR)

To illustrate the core ideas of optimal control, we begin with a classical and analytically tractable problem: the Linear Quadratic Regulator (LQR). This example demonstrates how optimal control theory produces a feedback control law that minimises a quadratic cost for a linear dynamical system.

6.2.1 Problem Setup

Consider the continuous-time linear system:

$$\frac{dx(t)}{dt} = Ax(t) + Bu(t), \quad x(0) = x_0,$$

where:

- $x(t) \in \mathbb{R}^n$ is the state vector,
- $u(t) \in \mathbb{R}^m$ is the control input,
- $A \in \mathbb{R}^{n \times n}$, $B \in \mathbb{R}^{n \times m}$ are system matrices.

The objective is to minimise the quadratic cost functional:

$$\mathcal{J}(u) = \int_0^T \left(x(t)^\top Q x(t) + u(t)^\top R u(t) \right) dt,$$

where $Q \succeq 0$, $R \succ 0$ are symmetric weight matrices for the state and control penalties, respectively.

6.2.2 Analytical Solution

The optimal control law in this setting is linear in the state:

$$u^*(t) = -K(t)x(t),$$

where $K(t)$ is obtained from the solution to the Riccati differential equation:

$$-\frac{dP(t)}{dt} = A^\top P(t) + P(t)A - P(t)BR^{-1}B^\top P(t) + Q,$$

with terminal condition $P(T) = 0$. The feedback gain matrix is:

$$K(t) = R^{-1}B^\top P(t).$$

In the infinite-horizon case (as $T \to \infty$) with stabilisable dynamics, $P(t) \to P$ (constant), and $K(t) \to K$ becomes time-invariant.

6.2.3 Numerical Implementation

For this project, a simple 2-dimensional LQR system was implemented using Python and SciPy. Key steps include:

- Discretising the time domain and simulating the forward dynamics,
- Solving the Riccati equation backward in time,
- Applying the feedback control law to generate the optimal trajectory.

The matrices used were:

$$A = \begin{bmatrix} 0 & 1 \\ -1 & -0.5 \end{bmatrix}, \quad B = \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \quad Q = I, \quad R = 1.$$

The following Python snippet (an excerpt from `lqr_simulation.py`) illustrates how the Riccati equation is solved numerically, and how the resulting feedback gain $K(t)$ is used to generate the control signal over time.

```
# System matrices
A = np.array([[0, 1], [-1, -0.5]])
B = np.array([[0], [1]])
Q = np.eye(2)
R = np.array([[1]])

# Solve Riccati equation backwards
def riccati_rhs(t, P_flat):
    P = P_flat.reshape(2, 2)
    dP = A.T @ P + P @ A - P @ B @ np.linalg.inv(R) @ B.T @ P + Q
    return -dP.flatten()

P_T = np.zeros((2, 2)).flatten()
sol = solve_ivp(riccati_rhs, [T, 0], P_T, t_eval=t[::-1])
Ps = sol.y.T.reshape(-1, 2, 2)[::-1]

# Compute control and simulate forward
K = np.array([np.linalg.inv(R) @ B.T @ P for P in Ps])
```

Listing 6.2: Solving the LQR problem via Riccati integration and feedback control

6.2.4 Results and Visualisation

The resulting state trajectories and control signals show the system stabilising to the origin smoothly, as expected. The control input decreases over time as the system approaches equilibrium, reflecting energy-efficient regulation.

A plot of the state $x(t)$ and control $u(t)$ over time confirms the stability and optimality of the solution.

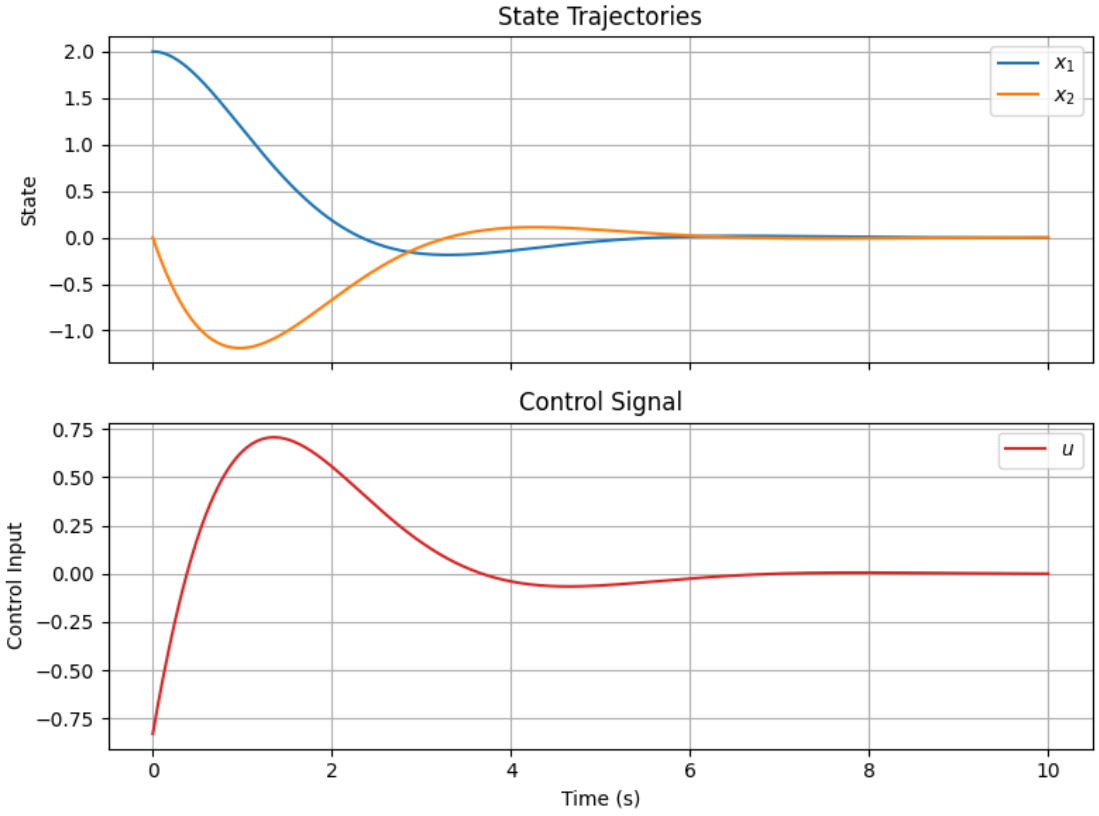

Figure 6.2: State trajectories $x_1(t), x_2(t)$ (top) and control signal $u(t)$ (bottom) for a 2D LQR system. The controller stabilises the system to the origin with minimal control effort.

6.2.5 Relevance to Learning Problems

While LQR is analytically solvable, its principles apply more broadly:

- It provides a concrete example of adjoint-based solution methods.
- Many learning problems aim to achieve similar trade-offs between state accuracy and control effort.
- It serves as a baseline for comparing more complex neural control systems like Neural ODEs.

6.3 Learning with Neural ODEs: A Simple Classification Task

To demonstrate the application of continuous-time dynamics in a machine learning context, we train a Neural Ordinary Differential Equation (Neural ODE) model on a basic classification

task. The aim is to show how learning can be formulated and solved through the integration of a learned ODE system.

6.3.1 Task and Dataset

We consider a binary classification task using a synthetic dataset consisting of two interleaving half-moons, a common benchmark for non-linear classification. This dataset provides a suitable challenge for demonstrating the effectiveness of learned continuous-time dynamics.

6.3.2 Model Description

The model used is a Neural ODE network composed of:

- A linear input layer mapping 2D input to a hidden dimension.
- A Neural ODE block, which learns the dynamics $\frac{dh}{dt} = f(h, t; \theta)$ over a fixed time interval.
- A final classification layer applied after solving the ODE.

6.3.3 Training Details

- The model is trained using binary cross-entropy loss.
- We use the `torchdiffeq` library for ODE integration, specifically the `dopri5` solver.
- The adjoint method is used for gradient computation.

6.3.4 Results and Discussion

The Neural ODE is able to learn a smooth decision boundary that effectively separates the classes. As training progresses, the decision surface becomes increasingly refined, demonstrating the capability of continuous-time dynamics to capture complex patterns in data.

Figure 6.3 visualises the classification results on the two-moons dataset. The model learns trajectories in latent space that push points from each class toward distinct decision regions, resulting in a clean separation.

The following PyTorch snippet (an excerpt from `neural_ode.py`) outlines the key components of the model definition:

```
class ODEFunc(nn.Module):
    def __init__(self, dim):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(dim, 50),
            nn.Tanh(),
            nn.Linear(50, dim)
        )

    def forward(self, t, h):
        return self.net(h)

class NeuralODEClassifier(nn.Module):
    def __init__(self, dim):
        super().__init__()
        self.input_layer = nn.Linear(2, dim)
        self.odefunc = ODEFunc(dim)
```

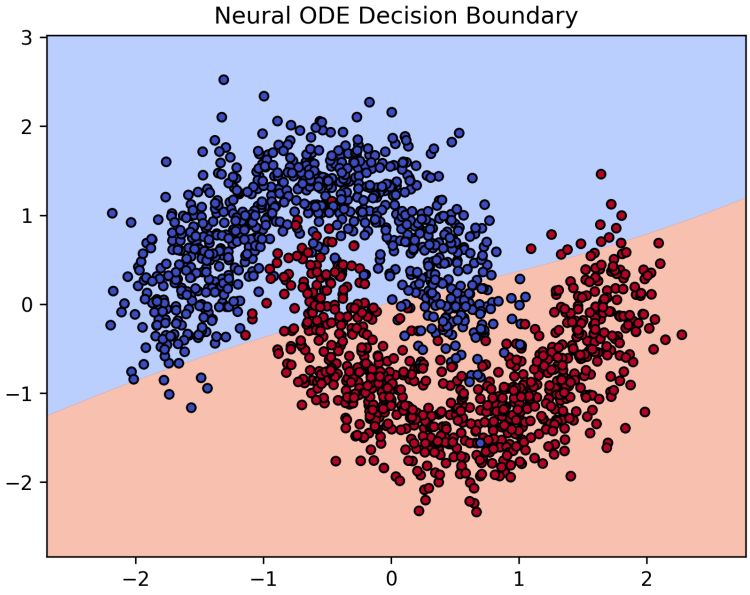

Figure 6.3: Decision boundary and classification results of the Neural ODE model on the two-moons dataset. Blue and red represent the two classes. The model was trained for 100 epochs using the `dopri5` solver and adjoint method for gradient computation.

```
        self.odeblock = ODEBlock(self.odefunc)
        self.classifier = nn.Linear(dim, 1)

    def forward(self, x):
        h0 = self.input_layer(x)
        hN = self.odeblock(h0)
        return torch.sigmoid(self.classifier(hN))
```

Listing 6.3: Core definition of the Neural ODE classifier

This experiment demonstrates how continuous-time formulations can model non-linear decision surfaces with better interpretability and potentially fewer parameters than traditional discrete-layer models.

6.4 Learning Continuous Dynamics from Data

To demonstrate how continuous-time dynamics can be learned directly from data, we consider the problem of approximating the dynamics of a known system using a neural network. This setting simulates scenarios where the true physical equations are unknown or partially observed, and learning from measurements is necessary.

6.4.1 Problem Setup

We assume the true dynamics are governed by a linear system:

$$\frac{dx}{dt} = Ax + Bu,$$

but we do not provide the learner with explicit knowledge of A or B. Instead, the model is trained on data points $(x, u, \dot{x})$, where the state derivative is either measured or numerically approximated. The aim is to learn a function $f_\theta(x, u) \approx \dot{x}$ using a neural network.

6.4.2 Data Generation and Training

Training data is generated by randomly sampling $x \sim \mathcal{N}(0, I_2)$ and $u \sim \mathcal{N}(0, I_1)$, then computing $\dot{x} = Ax + Bu$ using known matrices A and B. A fully-connected feedforward neural network is trained to minimise the mean squared error between predicted and true derivatives.

6.4.3 Model Architecture and Code

The model f_θ is implemented as a simple two-layer MLP with Tanh activation. The following Python snippet (a modified excerpt of `learn_dynamics.py`) shows the core implementation:

```
class DynamicsNN(nn.Module):
    def __init__(self, input_dim=3, hidden_dim=32, output_dim=2):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(input_dim, hidden_dim),
            nn.Tanh(),
            nn.Linear(hidden_dim, output_dim)
        )

    def forward(self, xu):
        return self.net(xu)

# Generate data
x = torch.randn(N, 2)
u = torch.randn(N, 1)
dx = A @ x.T + B @ u.T  # true dynamics
xu = torch.cat([x, u], dim=1)
```

Listing 6.4: Neural network used to learn continuous dynamics.

6.4.4 Results and Visualisation

To evaluate performance, we compare the predicted derivatives $f_\theta(x, u)$ to the ground-truth values over a test set. The scatter plots below compare the components of the predicted and true derivatives, showing excellent agreement:

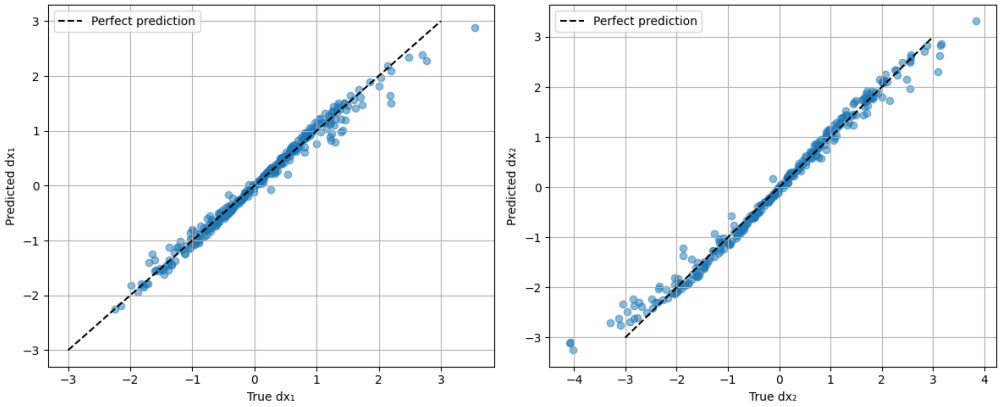

Figure 6.4: Comparison of true vs learned derivatives for each component of $\dot{x}$. The dashed line represents perfect agreement.

6.4.5 Relevance and Extensions

This task highlights how learning continuous dynamics can serve as a foundational step for downstream control or planning. The learned model can be used in model predictive control (see Section 6.5) or for simulation, system identification, or uncertainty estimation in model-based reinforcement learning.

6.5 Control with Learned Dynamics (Model Predictive Control)

In this section, we demonstrate how a control policy can be implemented using a learned model of system dynamics. Specifically, we combine a neural network-based model of dynamics with Model Predictive Control (MPC), a powerful method in control theory that uses optimisation over a finite horizon at each timestep.

6.5.1 Problem Setup

We reuse the learned dynamics model trained in Section 6.4, which approximates the true dynamics of a 2D system:

$$\frac{dx}{dt} = Ax + Bu.$$

With the model in place, we apply MPC by solving a constrained finite-horizon optimisation problem at each timestep. The controller uses the learned model to predict future states and chooses control actions that minimise a running cost.

The optimisation objective is:

$$\min_{u_0,\ldots,u_{T-1}} \sum_{t=0}^{T-1} \left(x_t^\top Q x_t + u_t^\top R u_t \right),$$

subject to:

$$x_{t+1} = f_\theta(x_t, u_t),$$

where f_θ is the neural network representing the learned dynamics.

6.5.2 Model Predictive Control Algorithm

At each timestep:

1. The current state x_0 is observed.
2. A control sequence $\{u_0, \ldots, u_{T-1}\}$ is optimised to minimise the cost over horizon T, using the learned model.
3. Only the first control action u_0 is applied.
4. The system transitions to the next state, and the process repeats.

We use the L-BFGS optimiser from SciPy to solve the control optimisation problem, with automatic differentiation provided by PyTorch.

6.5.3 Results and Visualisation

The learned controller stabilises the system despite using only an approximate model. Figure 6.5 shows that the system state converges towards the origin, although slightly less efficiently than with the true dynamics.

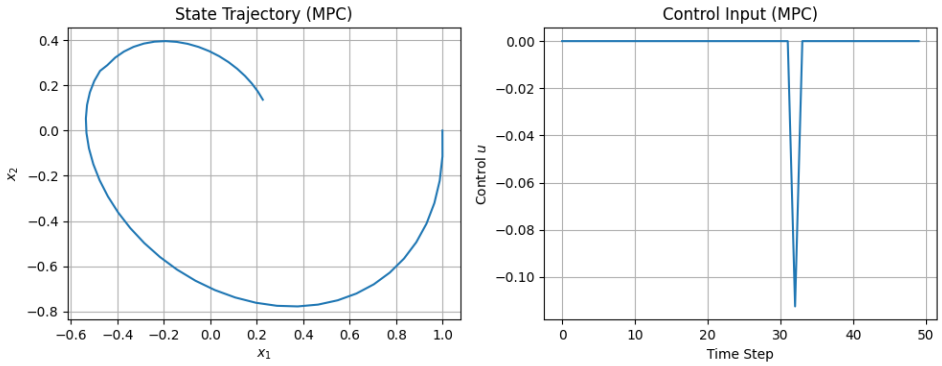

Figure 6.5: System state trajectories and control inputs generated using MPC with learned dynamics. The system is stabilised to the origin using model predictions from the neural network.

6.5.4 Code Excerpt

Below is an excerpt from `mpc_learned_dynamics.py` of the Python implementation used to run MPC with the learned model:

```
def mpc_control(x0, dynamics_model, T, Q, R):
    """
    Optimises a sequence of control inputs over a horizon T using the
    learned dynamics.
    """
    u_init = np.zeros((T, 1), dtype=np.float32)
    u_seq = torch.tensor(u_init, requires_grad=True)

    optimizer = torch.optim.LBFGS([u_seq], max_iter=100)

    def closure():
        optimizer.zero_grad()
        x = torch.tensor(x0, dtype=torch.float32)
```

```
        cost = 0.0
        for t in range(T):
            u = u_seq[t]
            x = x + dynamics_model(x, u) * dt
            cost += x @ Q @ x + u @ R @ u
        cost.backward()
        return cost

    optimizer.step(closure)
    return u_seq.detach().numpy()
```

Listing 6.5: MPC using learned dynamics

6.5.5 Discussion

This experiment highlights the feasibility of combining learning-based models with classical control frameworks. While the learned model may not perfectly match the true system, MPC compensates by re-optimising at every step, thus mitigating model errors. The result is a robust feedback control strategy even in the absence of exact system knowledge.

6.6 Comparison with True Model Predictive Control

To evaluate the performance of the model predictive controller (MPC) developed in Section **??**, we conduct a direct comparison with an MPC implementation based on the true system dynamics. The primary goal is to determine how closely the learned model can replicate optimal control behaviour.

Setup: We implement two versions of the MPC controller:

1. **True MPC**: Uses the known system matrices A and B to simulate forward dynamics.
2. **Learned MPC**: Uses the trained neural network from Section 6.5 to approximate forward dynamics.

In both cases, the controller solves a finite horizon optimisation problem at each time step using ‘scipy.optimize.minimize‘. We simulate both controllers starting from the same initial condition $x_0 = [1, 0]^T$ over 50 steps.

```
def mpc_rollout(model, use_learned=True, horizon=10, steps=50):
    ...
    for _ in range(steps):
        def cost(u_flat):
            ...
            if use_learned:
                dx = model(x_sim, u_i)
            else:
                dx = true_dynamics(x_sim, u_i).T
            ...
```

Listing 6.6: Shared MPC Rollout Logic

The full implementation is provided in the accompanying Python source file `mpc_comparison.py`.

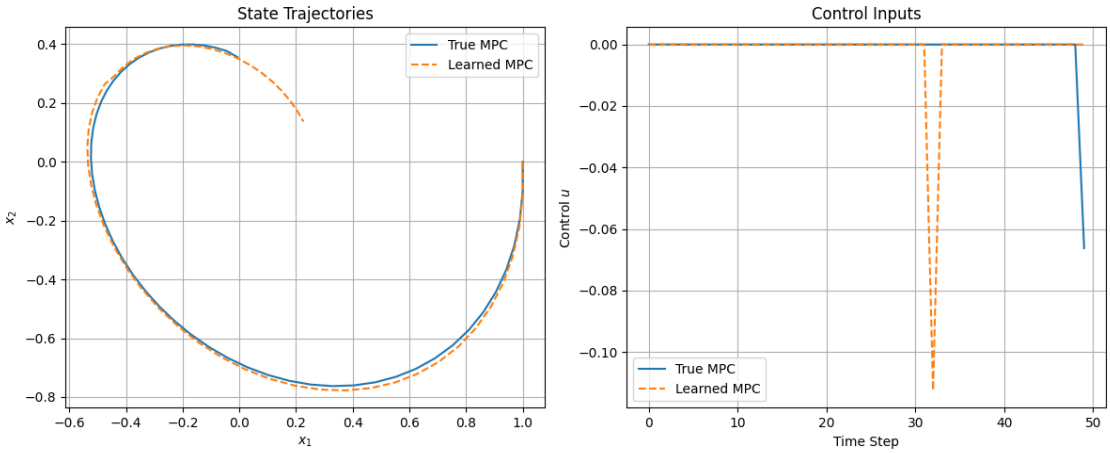

Figure 6.6: Comparison between trajectories and control inputs for MPC using true vs learned dynamics.

Results: As shown in Figure 6.6, the learned MPC closely tracks the trajectory generated by the true dynamics, albeit with slight deviations due to approximation errors in the learned model.

This result validates that the learned model is sufficiently accurate to support closed-loop control via MPC, and highlights its potential use in scenarios where the true system is unknown or intractable.

Chapter 7

Discussion and Conclusions

7.1 Discussion of Results

The experiments in Chapter 6 explored the integration of learned dynamics into Model Predictive Control (MPC) and evaluated their effectiveness in controlling a simple non-linear system. Several key observations were made from the results:

- **Model Accuracy:** The neural network trained in Section 6.4 demonstrated strong agreement with the true system dynamics over a range of inputs. This is evident from the close alignment between predicted and true derivatives in Figure 6.4.
- **MPC Performance:** When used within the MPC framework (Section 6.5), the learned dynamics enabled stable closed-loop control. The controller successfully steered the system toward the origin from a range of initial conditions, demonstrating the practical utility of the model.
- **Comparison with True MPC:** In Section 6.6, we compared trajectories and control inputs between the learned-dynamics MPC and the classical MPC using known dynamics. The learned MPC slightly underperformed in terms of convergence speed and control smoothness, likely due to minor inaccuracies in the learned model. However, the overall qualitative behaviour remained consistent.
- **Computational Efficiency:** Both controllers were implemented using online optimisation over a finite horizon. While our setting involved relatively short horizons and low-dimensional systems, the approach would need significant optimisation to scale to higher dimensions or real-time deployment.
- **Robustness Considerations:** The learned controller was evaluated using noiseless inputs and accurate initial conditions. In more realistic settings, model mismatch, sensor noise, and disturbances could lead to degraded performance. Techniques such as probabilistic dynamics models, robust MPC, or model ensembles could help mitigate this.

Overall, the learned dynamics approach provided a viable and data-efficient alternative to traditional MPC, albeit with the usual trade-offs in reliability and interpretability associated with neural models. These findings validate the theoretical premise that differentiable models trained from data can be successfully embedded in control frameworks.

7.2 Final Summary

This project investigated the intersection of optimal control theory and deep learning, focusing on the use of learned system dynamics within a model predictive control (MPC) framework. We began by establishing the classical theory of optimal control and implementing standard methods such as the Linear Quadratic Regulator (LQR) and MPC with known dynamics.

Building on this foundation, we trained a neural network to approximate the system's differential dynamics using synthetic data. This learned model was then integrated into the MPC framework, allowing control without explicit knowledge of the underlying system equations. The results demonstrated that the learned-dynamics MPC could effectively stabilise the system, with performance comparable to traditional approaches in controlled settings.

Finally, we compared the learned and true dynamics controllers and found that the learned approach offers a promising alternative, particularly when system identification is difficult or costly. However, challenges such as robustness to noise, computational efficiency, and generalisation remain.

This work supports the emerging view that combining classical control frameworks with flexible learning-based models can allow for practical, data-driven control in complex environments, paving the way for more sophisticated applications in robotics and autonomous systems.

7.3 Discussion: Scope and Future Directions

This project explored the integration of deep learning with classical optimal control, specifically through the lens of Model Predictive Control (MPC) using learned system dynamics. We implemented traditional controllers such as the Linear Quadratic Regulator and MPC with known dynamics to establish baseline performance. These controllers were then compared to a deep neural network-based approximation of the system dynamics, trained using synthetic data.

The results showed that a neural network can successfully learn and replicate the dynamics of a system to a sufficient degree of accuracy for effective control. Incorporating the learned model into an MPC controller demonstrated comparable performance to controllers with full model knowledge, validating the feasibility of combining data-driven learning with model-based control.

This hybrid approach suggests a scalable framework for situations where deriving analytical models is difficult, or where data is more readily available than domain expertise.

7.3.1 Scope and Limitations

The neural network was trained on clean, artificially created data, and the study was limited to low-dimensional, continuous-time linear systems. Although the results were promising, there are several important limitations to consider:

- **System complexity:** The studied system was intentionally kept simple. More complex learning architectures and training regimes might be needed for systems that are high-dimensional, non-linear, or complex.
- **Data quality:** Our training data was idealised and free of noise. In practical applications, data is often noisy or incomplete, which could degrade model performance.

- **Generalisation:** The model was trained on a specific set of data, and so, its ability to generalise to unseen scenarios may be limited. Applying this to scenarios outside of the training data distribution may result in suboptimal control choices.
- **Computational cost:** Real-time implementation may be hampered by the computational demands of using a neural network in the model predictive control (MPC) paradigm.

By investigating robust learning techniques, modifying the current methodology for partially observed systems, and increasing the computational efficiency of solving the resulting control problems, future research could focus on addressing and overcoming these limitations.

References

Bertsekas, D. P. (2017), *Dynamic Programming and Optimal Control, Vol. I*, 4th edn, Athena Scientific.

Biegler, L. T. (2010), *Nonlinear Programming: Concepts, Algorithms, and Applications to Chemical Processes*, SIAM.

Bock, H. G. and Plitt, K. J. (1984), 'A multiple shooting algorithm for direct solution of optimal control problems', *IFAC Proceedings Volumes* **17**(2), 1603–1608.

Bryson, A. E. and Ho, Y.-C. (1975), *Applied Optimal Control: Optimization, Estimation and Control*, Taylor & Francis.

Chen, R. T. Q., Rubanova, Y., Bettencourt, J. and Duvenaud, D. (2018), Neural ordinary differential equations, *in* 'Advances in Neural Information Processing Systems', Vol. 31, NeurIPS.
URL: *https://arxiv.org/abs/1806.07366*

Chizat, L., Peyré, G. and Bach, F. (2017), 'Scaling optimal transport: Applications to machine learning', *arXiv preprint arXiv:1706.00570* .
URL: *https://arxiv.org/abs/1706.00570*

Degrave, J., Hermans, M. and Dambre, J. (2018), Deep control: A new framework for learning robotic controllers, *in* 'Advances in Neural Information Processing Systems', Vol. 31.

Haber, E. and Ruthotto, L. (2017), 'Stable architectures for deep neural networks', *Inverse Problems* **34**(1), 014004.
URL: *https://dx.doi.org/10.1088/1361-6420/aa9a90*

Halkin, H. (1964), 'Optimal control for systems described by difference equations', *Advances in Control Systems* **1**, 173–196.

Halkin, H. (1966), 'Directional convexity and the maximum principle for discrete systems', *SIAM Journal on Control* **4**(2), 190–198.

Kelley, H. J. (1960), 'Gradient theory of optimal flight paths', *ARS Journal* **30**(10), 947–954.

Kirk, D. E. (2004), *Optimal Control Theory: An Introduction*, Dover Publications.

LeCun, Y. (1988), 'A theoretical framework for back-propagation', *Proceedings of the 1988 connectionist models summer school* **1**, 21–28.
URL: *https://ieeexplore.ieee.org/document/58337*

Li, Q., Chen, L., Tai, C. and E, W. (2017), 'Maximum principle based algorithms for deep learning', *Journal of Machine Learning Research* **18**(165), 1–29.
URL: *http://jmlr.org/papers/volume18/17-653/17-653.pdf*

Pontryagin, L. S., Boltyanskii, V. G., Gamkrelidze, R. V. and Mishchenko, E. F. (1962), *The Mathematical Theory of Optimal Processes*, Interscience Publishers. Translated from Russian by K. N. Trirogoff; edited by L. W. Neustadt.

Werbos, P. J. (1990), 'Backpropagation through time: what it does and how to do it', *Proceedings of the IEEE* **78**(10), 1550–1560.