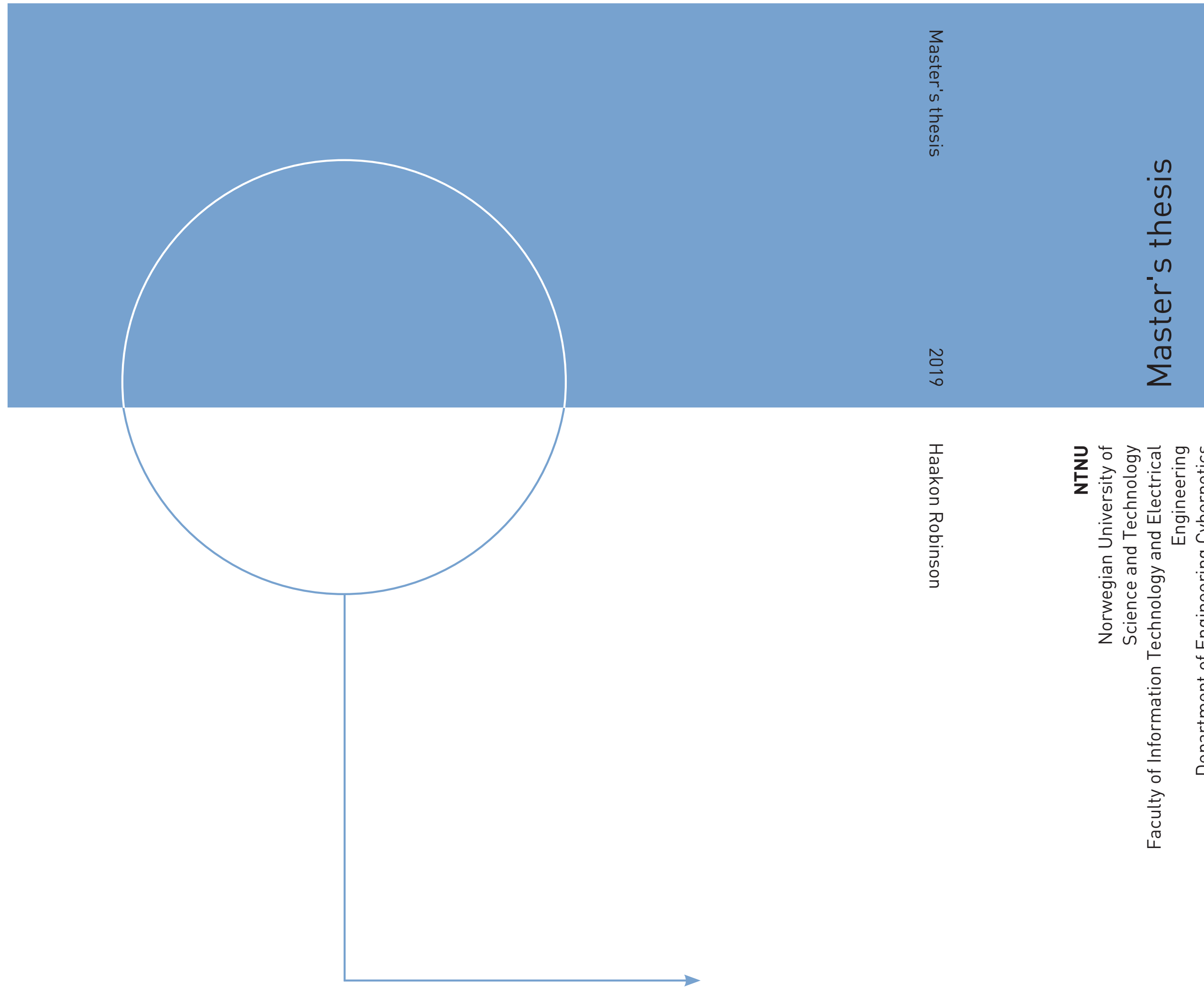



Haakon Robinson

# On the Piecewise Affine Representation of Neural Networks

## A Novel Approach to Explainable AI

June 2019

NTNU
Norwegian University of
Science and Technology

NTNU
Norwegian University of
Science and Technology

Norwegian University of
Science and Technology

# On the Piecewise Affine Representation of Neural Networks

A Novel Approach to Explainable AI

**Haakon Robinson**

MTTK
Submission date: June 2019
Supervisor: Professor Adil Rasheed

Norwegian University of Science and Technology
Department of Engineering Cybernetics

# Abstract

In exchange for large quantities of data and processing power, learning algorithms have yielded models that provide state of the art predication capabilities in many fields. However, the lack of strong guarantees on their behaviour have raised concerns over their use in safety-critical applications.

It can be shown that neural networks with piecewise affine (PWA) activation functions are themselves PWA, with their domains consisting of a vast number of linear regions. Research on this topic has focused on counting the number of linear regions, rather than obtaining the explicit PWA representation. This thesis presents a novel algorithm that can compute the PWA form of fully connected neural networks with ReLU activations. Several case studies regarding the usefulness of this representation in terms of modeling and control are undertaken. Nominal stability results for a simple dynamical system based on a small neural network are obtained via the Lyapunov method for PWA systems, and suggestions for extending the approach to neural networks of arbitrary size are outlined. Moreover, the practicality of using MPC and data-driven methods to control neural networks is investigated.

# Preface

The original premise for this Masters project/thesis was the application of machine learning techniques to problems in control theory, in particular marine motion control. Challenges related to model interpretability in the pre-project phase prompted me to explore the piecewise affine nature of neural networks alongside methods and insights gained from the specialisation course TTK18, held by Professor Morten Hovd. The result of that exploration is this thesis.

All of the presented algorithms and experiments were implemented in MATLAB using the MPT and YALMIP toolboxes. The MPT toolbox was especially useful for its library of polyhedral set operations and its ability to visualise collections of polyhedra.

All results were generated using a desktop computer, with a 6-core processor and 16 GB of RAM. It is likely that the work could have benefited from access to greater computational resources, as the developed methods turned out to be quite intensive in terms of computation and memory.

This thesis was written under the auspices of the Norwegian University of Science and Technology. Professor Adil Rasheed has been my advisor, and has assisted with the overall presentation of the thesis, proofreading, and helped put the work in a larger context. I am grateful for his enthusiasm, his encouragement, and his constructive feedback, which was invaluable. Professor Morten Hovd kindly advised me on the use of the MPT toolbox and the implementation of the algorithm. I'd also like to thank my family and friends for bringing me down to earth from time to time.

June 2019, Trondheim

Haakon Robinson

# Table of Contents

**1 Introduction** **1**

**2 Background** **2**
2.1 Related work and the state of the art . . . . . 2
2.2 Notation . . . . . 4
2.3 Theory . . . . . 4
2.3.1 Dynamical Systems and Control . . . . . 4
2.3.2 Optimisation . . . . . 6
2.3.3 Model Predictive Control . . . . . 8
2.3.4 Computational Geometry . . . . . 10
2.3.5 Piecewise Affine (PWA) Systems . . . . . 12
2.3.6 Neural Networks . . . . . 14

**3 Method and approach** **17**

**4 Algorithm: PWA Conversion** **19**
4.1 Derivation of the algorithm via examples . . . . . 19
4.1.1 Example 1: Simple network . . . . . 19
4.1.2 Example 2: Adding an output layer . . . . . 20
4.1.3 Example 3: Adding ReLU activation . . . . . 22
4.1.4 Example 4: Finding the regions . . . . . 23
4.2 Algorithm . . . . . 24
4.3 Optimisations . . . . . 26
4.4 Runtime . . . . . 28

**5 Case Study Results** **31**
5.1 Case study I: Simple LTI system . . . . . 31
5.2 Case study II: Hybrid model for Unforced Pendulum . . . . . 34
5.3 Case Study III: Applying MPC to a data-driven pendulum model . . . . . 38
5.4 Case Study IV: Closing the loop using neural network models and controllers . . . . . 42

**6 Discussion** **49**
6.1 Algorithm . . . . . 49
6.2 Case studies and future applicability of the work . . . . . 51

**7 Conclusion** **54**

**Bibliography** **55**

**Appendices** **57**

**A Methods in Computational Geometry** **58**

**B Algorithms for PWA systems** **59**
B.1 Computing the transition map for a PWA system . . . . . 59

B.2 Verifying that a set of polyhedral regions is positive invariant . . . . . . . . . . . . . . . . . . . 59
B.3 Finding the maximal positive invariant set . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 59

# 1 | Introduction

The past few years of research have been dominated by a modern-day, functional form of alchemy. In exchange for large quantities of data and processing power, learning algorithms have yielded models that provide state-of-the-art prediction capabilities in innumerable fields of research. Examples include the forecasting of stock prices for high speed trading, accurate predictions of protein folding, and playing all manner of games at a superhuman level. However, the lack of strong guarantees on their behaviour have raised concerns on their use in safety critical applications, particularly within engineering.

At the centre of this excitement lies *deep neural networks*, which have been proven to be so-called *universal approximators* (see Csáji (2001)). In other words, these networks can be scaled up and fitted to data of arbitrary complexity, even random noise, as shown in Raghu et al. (2017).

The main strength of these models, their flexibility, is also a danger. Reasoning about their behaviour, and explaining their output is notoriously difficult, leading to the term "black boxes". This makes it difficult to provide any performance or safety guarantees. There are already reports of fatal accidents. For example, a self-driving car hit and killed a pedestrian (Said (2018)), despite the logs showing that the pedestrian had been seen and recognised by the system. This has hindered adoption in safety-critical applications.

There is a need for new methods that can further the analysis of these models. Recent research has shown that neural networks that only use piecewise affine (PWA) activation functions can themselves be expressed as a PWA function defined on convex polyhedra, although the number of regions is enormous (Montufar et al. (2014); Pascanu et al. (2013); Raghu et al. (2017)).

However, there has been little research into explicitly finding and working with the PWA representation of neural networks. This is unfortunate, as there is a wealth of literature on PWA functions, particularly in the context of modeling and control. The premise of the work is to address this. The contribution of this thesis is:

- A **novel algorithm** that converts neural networks into an exact PWA representation that can be visualised and analysed.
- An investigation into the potential applications of the algorithm to modelling and control problems through a **series of case studies**.

The structure of the work is as follows. **Chapter 2** presents the guiding questions for this work and related research, as well as some preliminary comments on notation and relevant background theory on optimisation, $n$-dimensional polyhedra, nonlinear systems theory, PWA systems, and neural networks. **Chapter 3** fully formulates the problem, and discusses the approach taken to create the algorithm as well as how the case studies were designed. **Chapter 4** begins by motivating the design of the algorithm through the use of examples, and then presents the algorithm proper along with some optimisations and an analysis of its runtime complexity with respect to network size. **Chapter 5** presents four case studies in modeling and control, demonstrating the potential of the approach as well as some of the pitfalls. **Chapter 6** discusses the results, the limitations of the work, and potential avenues for further development. The conclusions of this thesis are presented in **Chapter 7**.

# 2 | Background

Three **guiding questions** were chosen to direct the focus of the research.

- *How can a neural network be converted to its piecewise affine form?*
- *How can the stability of a dynamical model based on a neural network be verified?*
- *Is it practical to design controllers for dynamical neural network models?*

The overarching aim of the work is then to find satisfactory answers to these questions.

## 2.1 | Related work and the state of the art

This thesis begins with the idea that neural networks using piecewise affine (PWA) activations are themselves PWA functions. This is shown in Eldan and Shamir (2015), along with a proof that large enough networks can approximate any PWA function on convex polyhedra. Note that this last point does not imply that the linear regions of the network can be *any* set of polyhedra, only that the output is the same.

Later studies of the linear regions of networks started with the need to understand how *expressive* a neural network is, and how this changes with its architecture (number of layers, width of layers, etc). A more expressive network has the ability to compute more complex, rich functions. Research has concluded that increasing the depth of a network has a bigger impact on expressivity than increasing the width of existing layers (Eldan and Shamir (2015); Telgarsky (2015)).

The premise of this research is that a network with many linear regions is more flexible/malleable. However, this is difficult to interpret, as the number of regions of a network will change as it undergoes training. Other expressivity measures have been developed, such as output trajectory length (Raghu et al. (2017)), however this does not relate directly to the linear regions and the PWA representation of neural networks.

Serra et al. (2018) presents upper and lower bounds on the maximum number of regions that improve on previous results (Montufar et al. (2014)), along with a mixed-integer formulation from which the regions can be counted by enumerating the integer solutions. For a network with input dimension $d$, $L$ hidden layers, each with $n$ nodes and ReLu activation, the asymptotic bounds for the *maximal* number of regions are:

$$\begin{aligned} \text{Lower: } & \Omega\Big( (\frac{n}{d})^{(L-1)d} n^d \Big) \\ \text{Upper: } & \mathcal{O}(n^{dL}) \end{aligned} \tag{2.1}$$

Note that the upper bound is exponential in both $d$ and $L$. The most challenging aspect in terms of analysis is that researchers are interested in networks with both large input dimension $d$ *and* many hidden layers (large $L$). The number of linear regions of such a network is enormous. It is likely due to this that (to the best of the authors knowledge) there are no previous works that detail methods to explicitly obtain and work with the PWA representation of neural networks.

A closely related problem is the enumeration of cells/regions and intersections in hyperplane arrangements. A hyperplane arrangement is simply a collection of full-dimensional hyperplanes in some space, such as a set of lines in $\mathbb{R}^2$. As will be shown in Chapter 4 (and is shown in the previous citations regarding the linear

regions of networks), there is a sense in which each hidden node in a neural network can be associated with a hyperplane, such that that the node is active on one side and inactive (clipped to zero by the ReLU function) on the other. The study of hyperplane arrangements is a rich topic, with some decades of results. A compact and rigorous discussion of hyperplane arrangements is given in Stanley (2006), while Zaslavsky (1975) gives an upper bound on the maximal number of regions for $n$ hyperplanes in $\mathbb{R}^d$:

$$\sum_{j=0}^{d} \binom{n}{j} \tag{2.2}$$

As with the bounds in (2.1), this expression grows quickly with both $d$ and $n$, but not exponentially.

Working with these objects in a numerical setting is difficult. The MPT toolbox for MATLAB (Herceg et al. (2013)) provides this functionality, as well as support for representing functions over sets, piecewise dynamical systems, and computing explicit MPC controllers, making it the obvious choice for the implementation. It relies on the equally useful optimisation toolbox YALMIP (Löfberg (2004)), which was also utilised. The MPT toolbox performs the majority of its set operations by solving LPs. These operations are covered conceptually in section 2.3.4.

There is good reason to be interested in neural networks within the context of modeling and control. Recently, some fascinating connections have been drawn between neural networks and differential equations. The expression for the hidden state in the successful *residual network* has a similar form to that of the typical *Euler method*: $y_{n+1} = y_n + hf(y_n, t_n)$. This seems to suggest that the network implements a rudimentary numerical integrator, which has previously motivated researchers to design residual network architectures that mimic more advanced numerical integrators like RK4, and also PDEs (Ruthotto and Haber (2018)). More interestingly, in the popular paper (Chen et al. (2018)) the authors were instead inspired to "parametrise the hidden state" of a neural network and use a numerical integrator to evaluate it. What this actually means in practice is that the *dynamics* of some system is learned by a neural network, and a an ODE solver is then applied to the output of the network. The real contribution of this paper is then a method to perform efficient backpropagation through the numerical solver. This allows the models to be trained on measured *trajectories* by simulating their dynamics and backpropagating the error over the whole trajectory.

This is not the first instance of a neural network being used to learn dynamics. In Leontaritis and Billings (1985), a model called the *Nonlinear AutoRegressive network with eXogenous inputs* (or NARXnets), was introduced, and further investigated in Siegelmann et al. (1997). Autoregressive models use a number of their previous outputs to predict the next output. They are often used to describe random processes, and are in fact very similar to multistep simulation methods. "Exogenous" just implies that some of the inputs are external, thus allowing the dynamic model to have control inputs. Being older, NARXnets have been studied more, and a scheme to use MPC with them is described in Chapter 8 of Grancharova and Johansen (2012).

While networks have been used to model dynamical systems, it should be noted that they do not follow typical physical conservation laws, and so their outputs may not be realistic. Efforts to correct this have focused on simulating scenes of objects by predicting movement and detecting collisions (see Grzeszczuk et al. (2000) and Chang et al. (2016)).

In addition to the PWA representability of neural networks and their ability to model dynamics, this thesis was also inspired by the field of explicit nonlinear MPC. In particular, the algorithms presented in Grancharova and Johansen (2012) outline how approximate solutions to the explicit MPC problem can be obtained for arbitrary nonlinear systems. The approach is to locally approximate the mp-NLP problem using mp-QPs over convex polyhedra. This is very related to the approximation of nonlinear functions using PWA functions, which is again related to neural networks. Grancharova and Johansen (2012) also dedicate a chapter to applying MPC to the previously mentioned NARXnets. Another work involving neural networks and MPC is Chen et al. (2018), where the explicit MPC control law is approximated by a PWA representable neural network using reinforcement learning. These links are discussed further in section 6.2.

## 2.2 | Notation

In general, the following conventions will be used.

- **Scalar values**, such as an element of some matrix, are written in lower case, with no special formatting. Examples: $a, b, c \dots$.
- **Vectors and matrices** are denoted with bold symbols. Matrices are capitalised ($\boldsymbol{W}$, $\boldsymbol{Z}$, $\boldsymbol{N}$), while vectors are not ($\boldsymbol{x}$, $\boldsymbol{y}$, $\boldsymbol{z}$).
- A vector $\boldsymbol{v}$ is always given as a **column vector**, so that $\boldsymbol{v}^T$ is unambiguously a **row vector**.
- **Discrete sets** are written with calligraphic font, i.e. $\mathcal{R}$, $\mathcal{W}$, $\mathcal{P}$, $\mathcal{N}$. When constructing a set of objects that are denoted with some letter, the set will be written as the calligraphic version of the same letter. For example, $\mathcal{P} = \{\boldsymbol{P}_i\}$.
- **Regions of some space** are written as unformatted upper case letters, i.e. $D \subset \mathbb{R}^{\ltimes}$. To differentiate these objects further, upper-case Greek letters such as $\Omega$ are used when appropriate.
- **Assignment** (in the context of **algorithms**) is written using a left arrow, i.e. $s \leftarrow 5$.
- **Concatenation of 2 discrete sets** is written as $A \parallel B$. This was chosen instead of the typical $Z = \{A, B, C\}$ (as done in MATLAB), because the latter could imply that $Z$ is a nested set of sets, rather than a flattened, concatenated set.

## 2.3 | Theory

### 2.3.1 | Dynamical Systems and Control

This section gives an overview of the terminology used when modeling dynamical systems and designing controllers. A more in-depth source on nonlinear systems is Khalil (2002).

#### State Equation

The state of a system represented as a vector $\boldsymbol{x}$. To express how this state evolves with time, a **state equation** is used, which is an Ordinary Differential Equation (ODE) in terms of $\boldsymbol{x}$:

$$\dot{\boldsymbol{x}} = f(t, \boldsymbol{x}, \boldsymbol{u}) \tag{2.3}$$

Here $t$ is the time, $\boldsymbol{x}$ the state, and $\boldsymbol{u}$ is an input vector that can be used to control the system. The properties of $f$ may be used to define a **control law**, or **controller**, $\boldsymbol{u} = \boldsymbol{u}(\boldsymbol{x})$ that generates an input value in response to measurements.

If the controller is chosen well, the state $x$ will converge to a desirable value. The choice of controller will often depend on what type of system $f$ is. If the system is linear, it has the form:

$$\dot{\boldsymbol{x}}(t) = \boldsymbol{A}(t)\boldsymbol{x}(t) + \boldsymbol{B}(t)\boldsymbol{u}(t) \tag{2.4}$$

This is a **linear time-varying system** (**LTV**). If the parameter time dependence is removed, then the system is **linear time-invariant** (**LTI**):

$$\dot{\boldsymbol{x}}(t) = \boldsymbol{A}\boldsymbol{x}(t) + \boldsymbol{B}\boldsymbol{u}(t) \tag{2.5}$$

LTIs are a well studied class of systems, and there are many effective control design strategies for them. Given a state equation and a control law, we can plug in for $u$ to obtain the **closed loop dynamics**:

$$\dot{\boldsymbol{x}} = f(t, \boldsymbol{x}, \boldsymbol{u}(\boldsymbol{x})) = f(t, \boldsymbol{x}) \tag{2.6}$$

This is an example of an **unforced system**. while (2.3) is a **forced** system. Removing the time dependence yields an **autonomous system**.

$$\dot{\boldsymbol{x}} = f(\boldsymbol{x}) \tag{2.7}$$

The properties of the closed loop system (2.6) can now be analysed to understand how the system will behave when the controller is connected to it. The concept of **stability** is especially important.

### Stability

Any $\boldsymbol{x}$ such that $\dot{\boldsymbol{x}} = \boldsymbol{0}$ is called an **equilibrium point**. Assume now that $\boldsymbol{x} = 0$ is an equilibrium point (the coordinates can always be shifted so that this is true). The equilibrium point can be stable, to varying degrees:

- **Unstable**: if not stable
- **Stable**: if $\forall\epsilon \geq 0$ there is a $\delta \geq 0$ such that $||x(0)|| \leq \delta \implies ||\boldsymbol{x}(t)|| \leq \epsilon, \;\; \forall t$
- **Asymptotically stable (AS)**: if stable *and* $||x(0)|| \leq \delta \implies \lim_{t\to\infty} \boldsymbol{x}(t) = \boldsymbol{0}$
- **Globally asymptotically stable (GAS)**: if $x(t) \to 0$ as $t \to \infty$ for any $x(0)$.

**Instability** implies that the state will diverge. In a real-life system this can cause unexpected behaviour and damage to the system/environment, and must be avoided. **Stability** by itself implies that given an initial state, there is a region that the state will never leave. For example, the Earths orbit around the Sun is for all intents and purposes stable. **AS** is a more desirable property, as it guarantees that the state converges to some value, as long as the initial state lies within some stable region. This is called the **region of attraction**. Finally, **GAS** implies that the region of attraction is infinitely large, containing all possible initial states.

Stability can sometimes be proven by studying the properties of the closed loop equation. In the case of the LTI $\dot{\boldsymbol{x}} = \boldsymbol{A}\boldsymbol{x}$, it is sufficient that the eigenvalues of $\boldsymbol{A}$ are strictly negative in order for the system to be GAS. This is not the case for nonlinear systems. As a simplification, a common approximation is to *linearise* nonlinear systems at their equilibrium points, such that they can be locally expressed as the linear system $\dot{\boldsymbol{x}} = \boldsymbol{A}\boldsymbol{x}$. If the linearised system is AS at $\boldsymbol{x} = 0$, then it can be concluded that the nonlinear system is also AS within some region around $\boldsymbol{x} = 0$. An estimate of the size of this region is then necessary.

### Lyapunov Method

A simple and elegant method for verifying the stability of a system is the **Lyapunov method**. The premise is that the energy of a stable physical system will tend to zero as time progresses. The Lyapunov method uses some strictly positive scalar function $V(\boldsymbol{x})$ in place of actual energy content, called the **Lyapunov function**.

The method states that the system is stable if there exists a Lyapunov function such that:

- $V(0) = 0$
- $V(\boldsymbol{x}) > 0, \;\; \forall \boldsymbol{x} \neq 0$
- $\dot{V}(\boldsymbol{x}) = \frac{\partial V}{\partial \boldsymbol{x}} f(\boldsymbol{x}) \leq 0$

If $\dot{V}(\boldsymbol{x}) < 0$ instead, then the system is AS.

A common choice for the Lyapunov function is $V(\boldsymbol{x}) = \boldsymbol{x}^T\boldsymbol{P}\boldsymbol{x}$, where $\boldsymbol{P}$ is a real symmetric matrix. For the linear system $\dot{\boldsymbol{x}} = \boldsymbol{A}\boldsymbol{x}$, then:

$$\dot{V}(\boldsymbol{x}) = \boldsymbol{x}^T(\boldsymbol{A}^T\boldsymbol{P} + \boldsymbol{P}\boldsymbol{A})\boldsymbol{x} \tag{2.8}$$

If a matrix $P$ can be found so that $\boldsymbol{A}^T\boldsymbol{P} + \boldsymbol{P}\boldsymbol{A}$ is negative definite, then according to the Lyapunov method the equilibrium point is AS. While this is not interesting for a linear system (recall that $\dot{\boldsymbol{x}} = \boldsymbol{A}\boldsymbol{x}$ is GAS if the eigenvalues of $\boldsymbol{A}$ are strictly negative), this argument can also be applied to nonlinear systems that can be approximated as a stable linear system around $x = 0$. The matrix $\boldsymbol{P}$ can then be used to estimate a region of attraction. A more rigorous treatment of the Lyapunov method can be found in Khalil (2002).

### 2.3.2 | Optimisation

This section explains what an optimisation problem is, and introduces the most important problem classes. For more detailed information about optimisation and various algorithms, see Nocedal and Wright (2006).

#### Overview

At its most general, optimisation is about finding a point $\boldsymbol{x}$ that is optimal in terms of some metric (maximum profit, minimum time, minimum area, etc) while satisfying a set of constraints (budget, fuel, etc). These constraints may be **equalities** (denoted by $\mathcal{E}$), or **inequalities** (denoted by $\mathcal{I}$). A general optimisation problem can be formulated as:

$$\min_{\boldsymbol{x}} \; f(\boldsymbol{x}) \tag{2.9}$$

$$\text{s.t.} \; c_i(\boldsymbol{x}) = 0, \quad \forall i \in \mathcal{E} \tag{2.10}$$

$$c_i(\boldsymbol{x}) \geq 0, \quad \forall i \in \mathcal{I} \tag{2.11}$$

The function (2.9) is called the **objective**, while (2.10) and (2.11) are the **constraints**. Note that if the maximum to some function is instead required, it can be negated: $\min -f(\boldsymbol{x})$.

Any potential solution to the problem must satisfy the constraints, i.e. it must be a **feasible point**. The set of all feasible points is the **feasible set**.

There may be multiple solutions to the problem, which are called **local solutions**. A **global solution** is the best solution among many, but is generally difficult to find except in some special cases (see next section). Various techniques can be used to get closer to the global optimum, such as simply running the optimisation algorithm many times at different starting points and selecting the best results.

#### Convexity

An important concept in optimisation is **convexity**. If an optimisation problem is convex, any local solution must be a global solution. This makes the problem significantly easier, and once the solution is found the search can be stopped as there is no better solution.

A problem is convex if:

(a) The objective function $f(\boldsymbol{x})$ is convex

(b) The feasible set is convex

A scalar function is convex if the following is true for *any* $\boldsymbol{x}$ and $\boldsymbol{y}$.

$$\alpha f(\boldsymbol{x}) + (1-\alpha) f(\boldsymbol{y}) \geq f(\alpha \boldsymbol{x} + (1-\alpha)\boldsymbol{y}), \quad \forall \alpha \in [0,1] \tag{2.12}$$

This can be understood as selecting any two points on the function and drawing a line between the two points. If the function itself always lies *beneath* the line, then it is convex. This gives it a natural "bowl shape", where there is a global minimum. Incidentally, if the opposite is true (the function value always lies above the line), then the function $f$ is **concave**, and $-f$ is convex.

A convex set has a similar property: if a line is drawn between any two points inside the set, then every point on the line must also lie within the set. This means that the set has no extremities. For the feasible set to be convex, the equalities (2.10) must be *linear* and the inequalities (2.11) must be concave.

#### Linear Programming (LP)

If the objective (2.9) and the constraints are linear functions, then the problem is a **linear program** (LP). These are extremely common, and are the most relevant for this thesis. The general formulation of an LP is:

$$\min_{\boldsymbol{x}} \quad \boldsymbol{c}^T \boldsymbol{x} \tag{2.13}$$

$$\text{s.t.} \quad \boldsymbol{A}\boldsymbol{x} = \boldsymbol{b} \tag{2.14}$$

$$\boldsymbol{x} \geq 0 \tag{2.15}$$

Note that the equalities are expressed as a matrix equation, while the inequalities have been replaced by $\boldsymbol{x} \geq 0$. In fact, any LP with linear constraints can be converted to this form.

All LPs are convex, as all the functions are linear. The objective function is a hyperplane, while the feasible set is a convex polyhedron (see section 2.3.4). The solution will be "pressed against" a subset of the inequality constraints, such some of the elements of $\boldsymbol{x}$ are zero[1]. These constraints are called **active**. Note that equality constraints are always active.

This is the basis of the successful **simplex method**, which makes a guess of which constraints are active, and then keeps switching them out until it reaches an optimum point. This can be seen as the algorithm hopping from vertex to vertex on the surface of the polyhedral feasible set. For these reasons, the simplex method belongs to a family of **active-set** nethods. The simplex method and its revisions are very effective for small to medium sized LPs, while the largest LPs are usually solved with interior point methods, which instead move towards the optimum through the bulk of the feasible set before moving along the surface towards the solution.

#### Quadratic Programming (QP)

Another important type of optimisation problem is the quadratic programming problem:

$$\min_{\boldsymbol{x}} \quad \frac{1}{2}\boldsymbol{x}^{\boldsymbol{T}} \boldsymbol{G} \boldsymbol{x}^T + \boldsymbol{c}^T \boldsymbol{x} \tag{2.16}$$

$$\text{s.t.} \quad \boldsymbol{a_i}^T \boldsymbol{x} = \boldsymbol{b_i}, \quad \forall i \in \mathcal{E} \tag{2.17}$$

$$\boldsymbol{a_i}^T \boldsymbol{x} \geq \boldsymbol{b_i}, \quad \forall i \in \mathcal{I} \tag{2.18}$$

[1] imagine rolling a ball down a uniform slope, until it hits the straight lines of the inequality constraints. The ball will nestle into one of the corners, or lie perfectly balanced on one of the edges.

Here $\boldsymbol{G}$ is a symmetric matrix. If $\boldsymbol{G}$ is a *positive semidefinite* matrix, then the problem is convex, and the problem is similar in complexity to an LP. QPs are often used when the square of certain variables needs to be minimised, which could correspond to the square error of some signal. For this reason, QPs are suitable for expressing some cost that needs to be minimised, i.e. in dynamic optimisation (see section 2.3.3).

#### Nonlinear Programming (NLP)

If an optimisation problem has a nonlinear objective function or some nonlinear constraints, then it is known as a **nonlinear program**. The problem can still be convex, as long as the objective is convex, the equality constraints linear, and the inequality constraints concave (see section 2.3.2). In general however, NLPs tend to be non-convex, and have the most general formulation:

$$\min_{\boldsymbol{x}} \; f(\boldsymbol{x}) \tag{2.19}$$

$$\text{s.t.} \; c_i(\boldsymbol{x}) = 0, \quad \forall i \in \mathcal{E} \tag{2.20}$$

$$c_i(\boldsymbol{x}) \geq 0, \quad \forall i \in \mathcal{I} \tag{2.21}$$

NLP solvers often locally approximate the problem with an easier one (such as LPs or QPs) at each step. The solutions to these subproblems can then are used as starting point for the next step, where a new local approximation is made. Despite being an approximation, this process allows the algortihm to converge to local optima when it is already sufficiently close. When the starting point is far from an optimum, other techniques must be used, although these are not discussed here. A very successful class of algorithms that solve QP subproblems at every timestep is called Sequential Quadratic Programming, or SQP.

#### Parametric Programming

The solution to parametric optimisation problems are themselves functions.

$$\begin{aligned} f^*(\theta) = \min_{\boldsymbol{x}} \;& f(\boldsymbol{x}, \theta) \\ \text{s.t.} \;& c(\boldsymbol{x}) = \mathbf{0}, \quad i \in \mathcal{E} \\ & c(\boldsymbol{x}) \geq \mathbf{0}, \quad i \in \mathcal{I} \end{aligned} \tag{2.22}$$

If $\theta$ contains several parameters, then the problem is referred to as a **multiparametric problem**, or **mp** problem for short. The optimal solutions are typically returned as piecewise functions, and the domains that they are valid on. This type of optimisation problem has been studied extensively in connection with Model Predictive Control.

### 2.3.3 | Model Predictive Control

Model predictive control (MPC) is a popular and effective family of control strategies that use online dynamic optimisation to generate control inputs. This section explains how dynamic optimisation problems are formulated, gives a brief look at MPC, and how explicit solutions to the problem can be obtained. For a more detailed examination of MPC, Rawlings and Mayne (2009) is an excellent resource.

### Dynamic optimisation

Given a discrete dynamic model $\boldsymbol{x}_t + 1 = f(\boldsymbol{x}_t, \boldsymbol{u}_t)$ of a system and a control objective, it is possible to frame the problem of selecting a suitable sequence of inputs as an optimisation problem. Assuming that the control objective is to set $\boldsymbol{x} = 0$, then the solution to the following problem minimises the "error" or "cost" of applying the input sequence:

$$\min_{\boldsymbol{u}} \sum_{t=0}^{N} \frac{1}{2} \boldsymbol{x}_{t+1}^T \boldsymbol{Q} \boldsymbol{x}_{t+1} + \boldsymbol{d}_x^T \boldsymbol{x}_{t+1} + \frac{1}{2} \boldsymbol{u}_t^T \boldsymbol{R} \boldsymbol{u}_t + \boldsymbol{d}_u^T \boldsymbol{u}_t \tag{2.23}$$

$$\text{s.t. } \boldsymbol{x_{t+1}} = f(\boldsymbol{x}_t, \boldsymbol{u}_t) \tag{2.24}$$

It is also possible to place constraints on the state and input. $N$ is called the **horizon** of the problem, as it controls how far ahead the method should look. If there are inaccuracies in the model, longer term predictions will less accurate, yielding bad results for large $N$. Equation (2.23) is also known as a **finite horizon optimal control** problem.

It is possible to formulate an **infinite horizon optimal control** problem by letting $N \to \infty$, and requiring that the sum in (2.23) converges to zero as $x \to 0$. The solution can then be obtained by solving the **algebraic Ricatti equation** (not given here), yielding the optimal control law:

$$\boldsymbol{u}(\boldsymbol{x}) = -\boldsymbol{K}\boldsymbol{x} \tag{2.25}$$

This controller is commonly known as the **linear quadratic regulator**, or **LQR**.

### Receding Horizon Method

Solving a finite horizon optimal control problem yields a sequence of inputs that can be applied to the system. However, in practice the model used in the optimisation problem will always differ slightly from reality, such that the longer term predictions will become increasingly unreliable.

Any practical controller must implement some kind of feedback loop. If the solution to (2.24) can be obtained efficiently, the problem can be solved repeatedly every timestep, and the first input in the solution sequence can then be applied to the system, discarding the rest. This is called the **receding horizon approach**, and forms the basis for a set of methods called **model predictive control** (MPC). The optimisation problem is usually made more efficient by linearising the model (2.24), yielding a **linear MPC** controller.

MPC has some advantages over traditional, hard-coded controllers. It is easy to set up and extend, and offers an acceptable degree of performance even before tuning as it always computes an optimal control input according to the objective function. It also easily facilitates bounds on the state and inputs, as these can be expressed as linear constraints in the optimisation problem. The disadvantages include the computational requirements, the need for a fairly accurate model, and a more complex stability analysis.

### Explicit MPC

While normal MPC methods solve the optimisation problem online, this may be intractable for cost-efficient microcontrollers. To get around this, the explicit solution to the dynamic optimisation problem can be computed and stored, such that the online computation is replaced by a simple lookup. This can be expressed as a mp-QP problem.

The explicit solution takes the form of a function that takes in any state and returns the corresponding first optimal input. This is challenging, because the dynamic optimisation problem must be solved for the

entire state space. However, it has been proven that a linear MPC controller can be expressed as a PWA function defined on polyhedral subregions (Bemporad et al. (2002)). This allows the optimal input to be found by searching through the polyhedra to find the one that contains the current state and computing the corresponding affine control law.

## 2.3.4 | Computational Geometry

This section gives a brief overview over the mathematical objects used in this work, how to represent them, and some of their properties. Fukuda et al. (2004) is an approachable resource on computational geometry and working with polyhedra.

### Hyperplanes and Half-Spaces

A hyperplane is the $n$-dimensional generalisation of lines and planes. The set of all points lying on a hyperplane can be described using a single (underdetermined) linear equation of the form:

$$\boldsymbol{w}^T\boldsymbol{x} = b \tag{2.26}$$

For example, a line in $\mathbb{R}^2$ would be $w_1x_1 + w_2x_2 = b$, while a plane in $\mathbb{R}^3$ becomes $w_1x_1 + w_2x_2 + w_3x_3 = b$[2]. In order to express a line in $\mathbb{R}^3$, another equation is needed, which corresponds to adding a row to $\boldsymbol{w}^T$ in (2.26)

The coefficients of (2.26) can be understood geometrically. The vector $\boldsymbol{w}$ is *normal* to the surface of the hyperplane, which determines the orientation. The equation can now be transformed slightly without affecting the hyperplane:

$$\frac{1}{||\boldsymbol{w}||_2}\boldsymbol{w}^T\boldsymbol{x} = \frac{b}{||\boldsymbol{w}||_2} \tag{2.27}$$

The term on the right hand side, $\frac{b}{||\boldsymbol{w}||_2}$, now describes the shortest (signed) distance from the origin to the hyperplane. This shows that if $b = 0$, the hyperplane must pass through the origin. If $b > 0$, then the normal $\hat{\boldsymbol{w}}$ will be pointing *away* from the origin, and vice versa. This can be visualised by first setting $b = 0$, allowing the hyperplane to pass through the origin. Then, the hyperplane is pushed a distance $\frac{b}{||\boldsymbol{w}||_2}$ (going backwards if $b < 0$ in the direction of its normal).

Note that this gives a certain *directionality* to a hyperplane. If the signs of (2.26) are flipped ($-\boldsymbol{w}^T\boldsymbol{x} = -b$), then the hyperplane will remain in place, but its normal will now point in the opposite direction. This has significance when hyperplanes are used in *inequalities*:

$$\left\{\boldsymbol{x} \mid \boldsymbol{w}^T\boldsymbol{x} \leq b\right\} \tag{2.28}$$

The regions of space defined by (2.28) is called a **half-space**, as the hyperplane essentially cuts $\mathbb{R}^d$ in two, leaving half the space.

[2] Note that if $w_3$ is set to zero in the latter equation, it would be the same to the former equation. However, the result would still be a plane in $\mathbb{R}^3$! This is because (2.26) is actually a system of equations (of 1 equation), to which the hyperplane is the solution space. $x_3$ is a free variable with no weight, so the solution space can have arbitrary $x_3$.

### Arrangements of Hyperplanes

When several hyperplanes in $\mathbb{R}^d$ are plotted together in an *arrangement*, they may intersect in all manner of ways, cordon off regions of space, and form overall interesting shapes.

Consider a collection of lines in $\mathbb{R}^2$. The enclosed regions between the lines will all be irregular polygons, while the unenclosed regions will stretch off to infinity. An important property of the regions is that none of the angles will be oblique (greater than 180°), also known as **convexity**. This will always be the case for a polygon defined using infinitely long lines, as trying to increase one of the angles by rotating one of the lines will just cut off more of the polygon.

Higher dimensional hyperplane arrangements are far more complex than line arrangements, as the hyperplanes can intersect in far more ways. For example, given three non parallel planes in $\mathcal{R}^3$, each pair of planes will intersect along a line. The three lines will then intersect at a single point, called a **vertex**. This is in contrast to $\mathbb{R}^2$, as all hyperplanes are just lines, and they can only intersect at points[3]. This trend continues in even higher dimensions, where 5-dimensional hyperplanes will intersect in 4-dimensional hyperplanes, which intersect as 3-dimensional hyperplanes, and so on.

### Convex Polyhedra

Convex polyhedra can be defined in terms of hyperplanes arrangements: a convex region of space bounded by hyperplanes. They may also be understood as the feasible set of a set of linear inequalities. These linear inequalities are called the **H-representation** of the polyhedron[4].

$$\left\{\boldsymbol{x} \mid \boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b}\right\} \tag{2.29}$$

Here, each row $\boldsymbol{A}_i^T$ of $\boldsymbol{A}$ and each element $b_i$ of $\boldsymbol{b}$ correspond to the half-space $\boldsymbol{A}_i^T\boldsymbol{x} \leq b_i$. Together, they describe a region of space. Note that the region may be empty!

The $\boldsymbol{A}$ matrix and $\boldsymbol{b}$ vector may be combined into a single matrix $\boldsymbol{H} = \begin{bmatrix}\boldsymbol{A} & \boldsymbol{b}\end{bmatrix}$. The advantage of this compactified H-representation is that every polyhedron can now be represented with a single matrix, where the number of columns is one more than the spatial dimension, and the number of rows corresponds to the number of constraints. The $\boldsymbol{H}$ matrix and the H-representation can therefore be considered synonymous, which the rest of this work will reflect.

### Equality Constraints

Lower dimensional polyhedra, such as a flat square in $\mathbb{R}^3$, may also be represented by adding equality constraints. This may be represented as an additional matrix, where each row corresponds to an equality. This matrix will be called the **He matrix**. However, any equality constraint can be expressed as the intersection of 2 inequality constraints. For example, the linear constraint $\boldsymbol{W}\boldsymbol{x} = \boldsymbol{b}$ may be expressed as:

$$\boldsymbol{W}\boldsymbol{x} \geq \boldsymbol{b} \;\; \cap \;\; \boldsymbol{W}\boldsymbol{x} \leq \boldsymbol{b} \tag{2.30}$$

Therefore, any set operation that can work with inequality constraints can easily be extended to equalities as well. Note that the equalities and inequalities are still usually kept separate, as many solvers for optimisation problems can use the equality constraints to simplify the optimisation problem.

[3]They can also intersect along a whole line, if two lines are exactly the same. This degenerate case is neglected.

[4]There is an equivalent representation called the V-representation, which consists of a set of vertices and rays. It is expensive to convert between the two, so it is normal to choose one. This thesis uses the H-representation only.

### Basic Set operations

Only convex polyhedra are H-representable. However, arbitrary shapes can be described by using collections of convex polyhedra. This comes with some challenges, such as how it can be ensured that all of the polyhedra in the collection remain disjoint, how to perform operations on the whole collection, etc. A list of the operations that are used in this work are given below. More detailed descriptions of each of these operations may be found in appendix A. An even more detailed summary of how to work with collections of convex polyhedra is given in Baotic (2009).

- Checking if a polyhedron is empty
- Intersection of 2 convex polyhedra
- Union of 2 convex polyhedra
- Set difference of 2 polyhedra ($P/Q$)
- Complement of a polyhedron
- Chebyshev centre of a polyhedron
- Removing redundant hyperplanes from the H-representation

## 2.3.5 | Piecewise Affine (PWA) Systems

Piecewise affine systems are a useful and flexible modeling tool. This section describes their description, and some basic methods. For more detailed examples of PWA systems and their stability properties, see Lazar (2006).

### Formulation

This thesis mainly considers discrete time PWA functions, which may be defined as:

$$\boldsymbol{x}_{t+1} = \boldsymbol{A}_i \boldsymbol{x}_t + \boldsymbol{f}_i, \quad \boldsymbol{x} \in \Omega_i \tag{2.31}$$

where $\boldsymbol{z}_t$ is the state at time $t$, $\Omega_i$ is some polyhedral region of the state space that has a corresponding transition matrix $\boldsymbol{A}_i$ and offset vector $\boldsymbol{f}_i$. The behaviour of the system changes depending on which region $\Omega_i$ the state is in. Despite being everywhere linear/affine, PWA functions can approximate any nonlinear system arbitrarily well, at the expense of adding more regions. These system are also sometimes called **switched systems**, which refers to the *switching* of the model dynamics as the state crosses from one region to another.

As in section 2.3.1, the coordinates are assumed to be shifted so that $\boldsymbol{z} = 0$ is an equilibrium point. This implies that for any region $\Omega_i$ that contains the origin, $\boldsymbol{f}_i = \boldsymbol{0}$.

If the system has input, then the state can be augmented such that $\boldsymbol{z} = \begin{bmatrix} x & u \end{bmatrix}^T$. This yields:

$$\boldsymbol{x}_{t+1} = \begin{bmatrix} \boldsymbol{A}_i & \boldsymbol{B}_i \end{bmatrix} \begin{bmatrix} \boldsymbol{x}_t \\ \boldsymbol{u}_t \end{bmatrix} + \boldsymbol{f}_i, \quad \begin{bmatrix} \boldsymbol{x}_t & \boldsymbol{u}_t \end{bmatrix}^T \in \Omega_i \tag{2.32}$$

Note that the region $\Omega_i$ now describes a region of space in the augmented *state-input* space. This is often simplified so that the regions only vary with the state, reducing the complexity of the system.

$$\boldsymbol{x}_{t+1} = \boldsymbol{A}_i \boldsymbol{x}_t + \boldsymbol{B}_i \boldsymbol{u}_t + \boldsymbol{f}_i, \quad \boldsymbol{x}_t \in \Omega_i \tag{2.33}$$

### Stability and Positive Invariant Sets

Given a dynamical system $\dot{x} = f(\boldsymbol{x})$, a set $\Omega$ is called **positive invariant** if:

- $\boldsymbol{x}(0) \in \Omega \implies \boldsymbol{x}(t) \in \Omega \forall t > 0$

In other words, if the system has its initial state within a positive invariant set, it will never leave that set[5]. The positive invariant set can therefore be seen as a region of stability (not asymptotic stability) for the system $\dot{x} = f(\boldsymbol{x})$. For piecewise systems, the positive invariant set becomes a set of regions that the system will never leave. The **maximal positive invariant set** is the largest possible set that is still positive invariant. Procedures for checking positive invariantness and finding the maximal invariant set are given in appendix B.

### Lyapunov method for PWA systems

The Lyapunov method (see section 2.3.1) can be applied to PWA systems. If the invariant set contains an equilibrium point, the point is AS if there is a valid Lyapunov candidate for the system over the positive invariant set. If there is no single Lyapunov candidate for all regions of the system, it may also be defined as a piecewise function. For continuous time systems, the Lyapunov function must still be continuous. However, this does not need to be the case when working with discrete time PWA systems! In this case, the Lyapunov function is instead required to decrease whenever the system transitions between regions. In order to verify this, all possible transitions between regions must be identified.

### Transition maps and reachable sets

This discussion is limited to autonomous discrete time PWA systems. Given the dynamics $\boldsymbol{x}_{t+1} = \boldsymbol{A}_i\boldsymbol{x}_t + \boldsymbol{f}_i$ and a region $\Omega_i$, all possible $\boldsymbol{x}_{t+1}$ can be computed as the affine map of $\Omega_i$. This is called the **forward reachable set**, and can also be expressed as:

$$\left\{\boldsymbol{x}_{t+1} = \boldsymbol{A}_i\boldsymbol{x}_t + \boldsymbol{f}_i \mid \boldsymbol{x}_t \in \Omega_i\right\} \tag{2.34}$$

where $\boldsymbol{A}_i$ and $\boldsymbol{f}_i$ are the system matrix and offset vector corresponding to region $\Omega_i$. The **backward reachable set** can also be found, as the inverse affine map of $\Omega_i$. It can be understood as the set of all states that could have reached $\Omega_i$.

$$\left\{\boldsymbol{x}_t \mid \boldsymbol{x}_{t+1} = \boldsymbol{A}_i\boldsymbol{x}_t + \boldsymbol{f}_i, \boldsymbol{x}_{t+1} \in \Omega_i\right\} \tag{2.35}$$

However, computing the backwards reachable set is more complicated when working with a PWA function, as the state may have started in a different region $\Omega_j$, where the dynamics $(\boldsymbol{A}_i, \boldsymbol{f}_i)$ are different.

To simplify calculations, the **transition map** is defined. The map consists of a square adjacency matrix $\boldsymbol{T}$ that contains all possible transitions from a source regions $\Omega_i$ to another region $\Omega_j$, and a collection of the subregions $\mathcal{R}$ where the transitions occur. The elements of the matrix $\boldsymbol{T}$ can be expressed as:

$$T_{ij} = \begin{cases} 1 & \text{if } \exists\, \Omega_{ij} \subset \Omega_i, \;\; \boldsymbol{x} \in \Omega_{ij} \implies \boldsymbol{A}_i\boldsymbol{x} + \boldsymbol{f}_i \in \Omega_j \\ 0 & \text{otherwise} \end{cases} \tag{2.36}$$

The elements of $\mathcal{R}$ are then the subregions $\Omega_{ij}$. Note that a region may transition to itself! A procedure for computing the transition map is given in the appendix B. More details may be found in the documentation of the MPT toolbox for MATLAB (Herceg et al. (2013)).

[5] Note that this is different from the concept of a *region of attraction*, as the latter implies that the state will converge to an equilibrium point.

### 2.3.6 | Neural Networks

#### Formulation

A neural network is an idealised computational model inspired by the behaviour of neurons in the brain. A basic neural network is composed of many nodes arranged in a series of layers. Typically the layers are fully connected, such that there is a weighted edge between every node in a layer and the next layer, as can be seen in figure 2.1. The intermediate layers are called **hidden layers**.

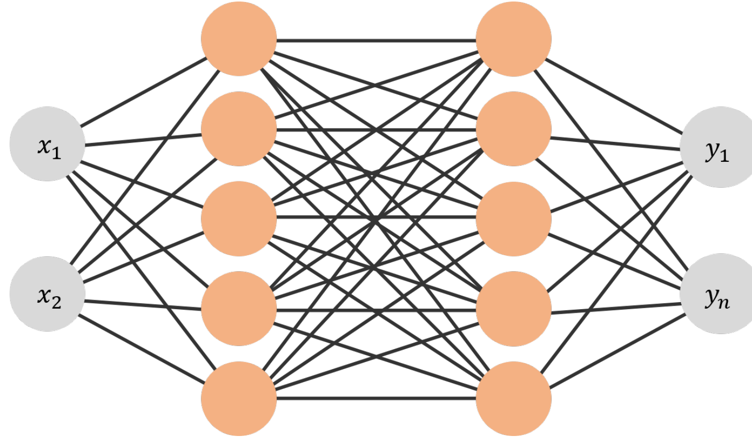



**Figure 2.1:** Example of a neural network with 2 input nodes, 2 output nodes, and 2 hidden layers with 5 nodes each. The diagram is coloured such that the orange nodes have an activation function, while the grey nodes do not. This colour scheme will be followed throughout the thesis.

The signals pass through the network from left to right, passing along the weighted edge to the next node. All of the inputs entering a node are then added. Nodes will also have a **bias** term that is added to their input. This is sometimes indicated in figures as an additional edge coming down from above, though this has not been shown here for clarity.

The connections between layers is equivalent to a matrix multiplication, while the bias terms are just an addition. This implies that each layer performs an affine transformation of the form $\boldsymbol{W}(\cdot) + b$ on the output of the previous layer. Fortunately, the result of two affine transformations a single affine transformation:

$$\begin{aligned} &\boldsymbol{W_n}(\cdots(\boldsymbol{W_2}(\boldsymbol{W_1}\boldsymbol{x} + \boldsymbol{b_1}) + \boldsymbol{b_2})) + \boldsymbol{b_n} \\ &= \boldsymbol{W_n}\cdots\boldsymbol{W_2}\boldsymbol{W_1}\boldsymbol{x} + (b_n + \cdots + \boldsymbol{W_n}\cdots\boldsymbol{W_2}\boldsymbol{W_1}b_1) \end{aligned} \tag{2.37}$$

Crucially, some of the layers may have an **activation function** (drawn in orange in the figures). This prevents the network from simplifying, and is what gives neural networks their representational power. The activation function for layer $i$ is written as $\sigma_i$. The network can now be written:

$$\Big(\sigma_n \circ (\boldsymbol{W}_n(\cdot) + \boldsymbol{b}_n) \circ \cdots \circ \sigma_2 \circ (\boldsymbol{W}_2(\cdot) + \boldsymbol{b}_2) \circ \sigma_1 \circ (\boldsymbol{W}_1(\cdot) + \boldsymbol{b}_1)\Big)(\boldsymbol{x}) \tag{2.38}$$

To compact the notation further, the weights and bias of each layer are placed into a homogeneous transformation matrix:

$$\boldsymbol{P}_k = \begin{bmatrix} \boldsymbol{W}_k & \boldsymbol{b}_k \\ \boldsymbol{0} & 1 \end{bmatrix} \tag{2.39}$$

This is called this the **parameter matrix** of layer $k$. This will make later operations much easier to express. Relaxing the notation a little, the composition operator is applied to matrix multiplication such that: $(\boldsymbol{P}_2 \circ \boldsymbol{P}_1)\boldsymbol{x} = \boldsymbol{P}_2\boldsymbol{P}_1\boldsymbol{x}$. The network can now be expressed as a series of alternating matrix multiplications and activation function applications.

$$\Big(\sigma_n \circ \boldsymbol{P}_n \circ \cdots \circ \sigma_2 \circ \boldsymbol{P}_2 \circ \sigma_1 \circ \boldsymbol{P}_1\Big)\left(\begin{bmatrix} \boldsymbol{x} \\ 1 \end{bmatrix}\right) \tag{2.40}$$

Note that a 1 has been concatenated to the state vector. This notation is similar to the way that neural networks are typically implemented; as long strings of operations, also known as computational graphs. The advantage of this is highlighted by (2.40); neural networks are very susceptible to the *chain rule*. This allows them to be easily differentiated in terms of their parameters, which makes them easy to optimise and fit to a dataset.

### Training Neural Networks

The training of a neural network is an iterative process where the network is presented with a data point, graded according to some **loss function**, and then updated based on the difference between its prediction and the expected value from the dataset. This is known as **supervised learning**, as it relies on the dataset having an output for every piece of data that it contains, although supervised learning encompasses more than just neural networks.

A common loss function is simply the squared error between the networks prediction $\hat{\boldsymbol{y}}$ and the actual output $\boldsymbol{y}$.

$$L(\boldsymbol{y}, \hat{\boldsymbol{y}}) = ||\boldsymbol{y} - \hat{\boldsymbol{y}}||_2^2 \tag{2.41}$$

Then, the parameters of the network can be changed through the use of an update law, also known as an **optimiser**. The simplest is gradient descent.

$$\boldsymbol{P}_i \leftarrow -\alpha \frac{\partial L}{\partial P_i} \tag{2.42}$$

Here $\alpha$ is the **learning rate**, and it determines how large the update should be. $\alpha$ is usually set quite low, typically around 0.001. Other popular optimisers include Stochastic Gradient Descent with Momentum (SGDM), RMSProp, and ADAM. These add *adaptive learning rates* and *momentum terms* to the base gradient descent algorithm for improved convergence. ADAM is considered by many to be the most effective optimiser.

In order for the network to reach an acceptable level of performance, it must typically pass over the entire dataset multiple times. Each pass over the dataset is called an **epoch**.

### Activation Functions

Any nonlinear function can in theory be used as an activation function, and there are a wide variety. One choice is $\tanh x$, as it maps all $x$ into the range $[-1, 1]$, making it very well behaved.

In this thesis, only the Rectifying Linear Unit, or **ReLU**, is considered. This is not necessarily a limitation, as ReLU is one of the most commonly used activation functions.

$$\text{relu}(x) = max(0, x) \tag{2.43}$$

Its behaviour is simply to clip any negative inputs to zero. It is efficient to implement, and it mitigates the **vanishing gradient** and **exploding gradient** problems[6] that affect larger networks.

Using ReLU can cause problems as well. During training, the parameters of a node might be adjusted in such a way that its output will always be negative, so that ReLU will then clip it to zero. This means that its parameters will never be updated, and it will remain "dead". This is known as the **dead neuron** problem. Leaky ReLU is a similar activation function that avoids dead neurons.

$$\text{leaky}(x) = \max(\alpha x, x) \tag{2.44}$$

where $\alpha << 1$. This gives the function a small gradient for negative x, allowing the error signal to pass through during backpropagation.

[6]The deeper a network is, the further the backpropagation process has to propagate the error. As mentioned, this is done using the chain rule. The gradient of the activation function in each layer thus works as a multiplier: if it is below unity the error signal will vanish, and vice versa. This makes it difficult to train the earlier layers of a network. ReLU's gradient is 1 if $x > 0$, and 0 otherwise.

# 3 | Method and approach

Referring back to the guiding questions outlined at the start of section 2, there are two main issues that need to be addressed:

(a) A practical, reasonably efficient algorithm is needed to perform the conversion from neural network to PWA function.

(b) It is unclear whether the resulting PWA functions will be practical for analysis. This should be investigated further.

The MPT toolbox for MATLAB (see Herceg et al. (2013)) provided the necessary computational geometry features for the algorithm in (a). An overview of the set operations that it provides are given in section 2.3.4. The software also supports PWA dynamical systems, MPC control, explicit MPC, and searching for Lyapunov functions, which was heavily utilised for the case studies. The MPT toolbox itself depends on the YALMIP toolbox (Löfberg (2004)), which acts as a layer between MATLAB users and a variety of optimisation solvers.

The PWA conversion algorithm was implemented entirely in MATLAB. Because of this, the algorithm was written to parse neural networks that have been defined using the Deep Learning Toolbox from Mathworks. This is seen as a compromise, as the most popular deep learning frameworks are written in Python. Since September 2017 it has been possible to transfer neural network model across platforms by using the Open Neural Network Exchange (ONNX) standard [1]. However, there still remain some challenges and bugs (see the discussion in 6.1. Therefore, in order to simplify matters as much as possible, everything was implemented in MATLAB and converting ONNX models was left as future work.

Once the algorithm was complete, some preliminary investigations into its practical use had to be undertaken. The case studies were designed with variety in mind, and were created by considering a broad overview of possible combinations of models and controllers (see table 3.1). Research on control synthesis for PWA systems has been predominantly focused on MPC and energy-based methods. Controllers are also sometimes *learnt*, as in reinforcement learning or behavioural cloning. For brevity, only MPC and data-driven (which are assumed PWA-representable) methods are considered.

The following cases (in order of complexity) were investigated:

**Case I**: As a proof of concept, a LTI system with a neural network controller was converted to its PWA form and analysed using the Lyapunov method.

**Case II**: This case study applies the same procedure as Case I to a more complex system, with the aim of seeing how the methods handle increased complexity. The system was selected to be a pendulum, as it exhibits simple nonlinear behaviour. Importantly, it has 2 states, making it possible to visualise the dynamics. A neural network was used to learn the dynamics of an unforced damped pendulum through a process of random sampling. The network is then converted to a PWA system, and the stability of the system is investigated using the Lyapunov method.

**Case III**: It is of interest to see how MPC techniques can be applied as-is to neural network models. This would reduce the diffculty of designing controllers for neural network models, and give the closed loop system desirable stability properties. Extending case II, a network is used to learn the dynamics

[1] Read more about ONNX here: `https://github.com/onnx/onnx`

of a forced pendulum. The system is then connected to an MPC controller, with the aim of inverting the pendulum.

**Case IV**: This case considers the possiblity of using neural network models and controllers in a closed-loop, and investigates how this affects the PWA representation of the closed-loop. Neural network controllers as a result of reinforcement learning or genetic optimisation are already used in some applications. The system under consideration is a fictional nonlinear spring. A neural network is then trained on data describing the force characteristic of the spring.

| | | **controller** | | |
|---|---|---|---|---|
| | | none | MPC | Data-Driven |
| **model** | LTI | - | - | Case I |
| | NL | Case II* | Case III* | Case IV* |
| | Data-driven | Case II | Case III | Case IV |

**Table 3.1:** Overview of possible combinations of models and controllers, and where this is investigated. *These cases are investigated approximately, as non-linear systems can be approximated arbitrarily well by PWA systems and can be lumped with data driven models.

# 4 | Algorithm: PWA Conversion

Before a formal introduction of the algorithm, a step-by-step, example driven overview of the process of converting a simple network to its PWA is presented. This is done in order to build an intuitive understanding of the inner working of the network as well as its transformation to its PWA representation. Although the discussion is limited to networks that use the popular ReLU activation function, the developed methods can be extended to networks with any piecewise linear/affine activation functions, i.e. Leaky ReLU or maxout (see Pascanu et al. (2013)). After the examples, the main algorithm is presented along with some optimisations. The chapter concludes with a quantitative evaluation of the computational runtime associated with the algorithm and its subroutines.

## 4.1 | Derivation of the algorithm via examples

### 4.1.1 | Example 1: Simple network

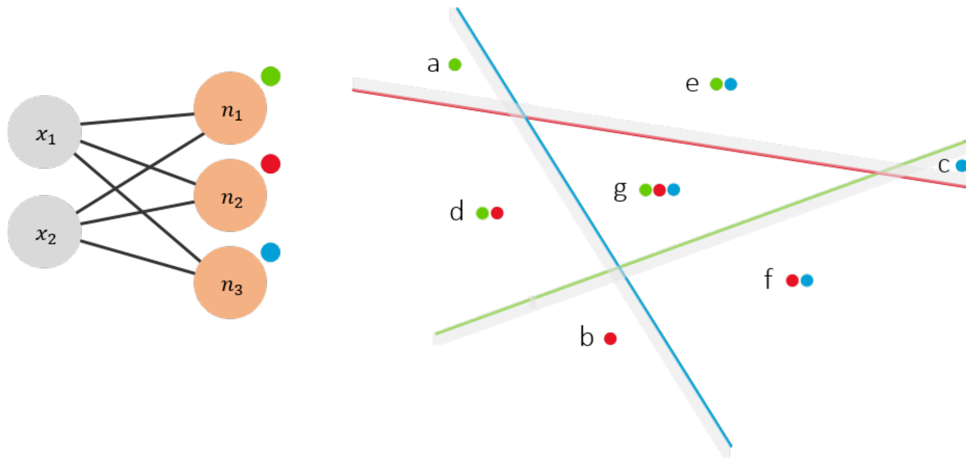



**Figure 4.1:** Simple network with 2 inputs and 1 hidden layer with 3 nodes. Each node defines a hyperplane that bisects the input space (in this case [2]). The activation pattern corresponding to each of the 7 regions is shown as a colour code.

First consider a neural network with 2 inputs and 1 hidden layer with 3 nodes and ReLU activation (see figure 4.1). The output layer is neglected for now, because it will not affect the linear regions of the network (see next example). The compact neural network notation from the preliminaries (see section 2.3.6) can be used to express the network as:

$$f(x) = \sigma(\boldsymbol{P}\bar{\boldsymbol{x}}) = \sigma\left(\begin{bmatrix}\boldsymbol{W} & \boldsymbol{b}\\ 0 & 1\end{bmatrix}\begin{bmatrix}\boldsymbol{x}\\ 1\end{bmatrix}\right) \tag{4.1}$$

where $\boldsymbol{P}$ is the parameter matrix that describes the connection weights between the input and the hidden layer. The activation function $\sigma$ is ReLU, which is a vector function that operates on each element of a column vector, clipping negative values at zero: $\sigma(x) = \max(0, x)$. Equation (4.1) can then be written as:

$$f(x) = \sigma\left( \begin{bmatrix} \boldsymbol{W} & \boldsymbol{b} \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \boldsymbol{x} \\ 1 \end{bmatrix} \right) = \begin{bmatrix} \max(0, \boldsymbol{w}_1^T \boldsymbol{x} + \boldsymbol{b}_1)) \\ \max(0, \boldsymbol{w}_2^T \boldsymbol{x} + \boldsymbol{b}_2)) \\ \max(0, \boldsymbol{w}_3^T \boldsymbol{x} + \boldsymbol{b}_3)) \\ 1 \end{bmatrix} \tag{4.2}$$

The vector $\boldsymbol{w}_i^T$ represents the $i^{\text{th}}$ row of $\boldsymbol{W}$. Each row corresponds to the output of a node. Each row/node has two modes: one where the output is clipped to zero (because $\boldsymbol{w}^T \boldsymbol{x} + \boldsymbol{b} \leq 0$), and one where the output is $\boldsymbol{w}^T \boldsymbol{x} + \boldsymbol{b}$. The boundary between these two modes is given by $\boldsymbol{w}^T \boldsymbol{x} + \boldsymbol{b} = 0$, which defines a hyperplane. Each node thus bisects the input space, only outputting a positive, nonzero value if the point x is on the positive side of the corresponding hyperplane. To visualise this directionality, hyperplanes will be drawn with a shaded side, as can be seen in figure 4.2.

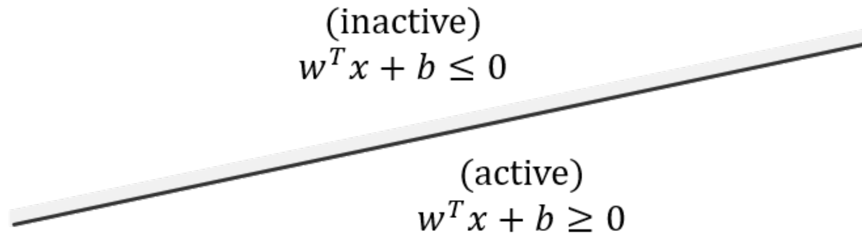



**Figure 4.2:** Each node with ReLU activation has two modes: one where it is active and one where it is inactive. Hyperplanes are drawn with a shaded side representing the inactive side.

The arrangement of these hyperplanes defines a set of polyhedral regions in the input space, each corresponding to a different set of active nodes. This is referred to as the **activation pattern**. Applying the ReLU function results in the negative elements of the output being clipped to zero. The effect of the ReLU nonlinearity is therefore better described as a *deactivation function*, although this term will not be used to avoid confusion. From (4.1) it can be seen that the *deactivation* of a node is equivalent to setting the corresponding row in the $\boldsymbol{P}$ matrix to zero. The function in each region is therefore described by its own copy of the $\boldsymbol{P}$ matrix, but with inactive rows being set to zero. This is shown explicitly for the simple network in table 4.1. As there are no further layers, this is the complete PWA representation of the simple network in figure 4.1. Note that the activation function $\sigma(\cdot)$ has been completely removed from the expression.

## 4.1.2 | Example 2: Adding an output layer

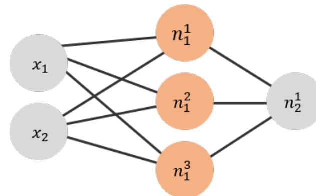



**Figure 4.3:** Adding another layer with no activation will not effect the linear regions at all. However, it will modify the $\boldsymbol{P}$ matrix of each region defined by the previous layers.

| Region | Corresponding Function |
|---|---|
| a (green) | $f_a(\boldsymbol{x}) = \begin{bmatrix} \boldsymbol{w}_1^T & b_1 \\ \mathbf{0} & 0 \\ \mathbf{0} & 0 \\ \mathbf{0} & 1 \end{bmatrix} \begin{bmatrix} \boldsymbol{x} \\ 1 \end{bmatrix}$ |
| b (red) | $f_b(\boldsymbol{x}) = \begin{bmatrix} \mathbf{0} & 0 \\ \boldsymbol{w}_2^T & b_2 \\ \mathbf{0} & 0 \\ \mathbf{0} & 1 \end{bmatrix} \begin{bmatrix} \boldsymbol{x} \\ 1 \end{bmatrix}$ |
| c (blue) | $f_c(\boldsymbol{x}) = \begin{bmatrix} \mathbf{0} & 0 \\ \mathbf{0} & 0 \\ \boldsymbol{w}_3^T & b_3 \\ \mathbf{0} & 1 \end{bmatrix} \begin{bmatrix} \boldsymbol{x} \\ 1 \end{bmatrix}$ |
| d (green, red) | $f_d(\boldsymbol{x}) = \begin{bmatrix} \boldsymbol{w}_1^T & b_1 \\ \boldsymbol{w}_2^T & b_2 \\ \mathbf{0} & 0 \\ \mathbf{0} & 1 \end{bmatrix} \begin{bmatrix} \boldsymbol{x} \\ 1 \end{bmatrix}$ |
| e (green, blue) | $f_e(\boldsymbol{x}) = \begin{bmatrix} \boldsymbol{w}_1^T & b_1 \\ \mathbf{0} & 0 \\ \boldsymbol{w}_3^T & b_3 \\ \mathbf{0} & 1 \end{bmatrix} \begin{bmatrix} \boldsymbol{x} \\ 1 \end{bmatrix}$ |
| f (red, blue) | $f_f(\boldsymbol{x}) = \begin{bmatrix} \mathbf{0} & 0 \\ \boldsymbol{w}_2^T & b_2 \\ \boldsymbol{w}_3^T & b_3 \\ \mathbf{0} & 1 \end{bmatrix} \begin{bmatrix} \boldsymbol{x} \\ 1 \end{bmatrix}$ |
| g (green, red, blue) | $f_g(\boldsymbol{x}) = \begin{bmatrix} \boldsymbol{w}_1^T & b_1 \\ \boldsymbol{w}_2^T & b_2 \\ \boldsymbol{w}_3^T & b_3 \\ \mathbf{0} & 1 \end{bmatrix} \begin{bmatrix} \boldsymbol{x} \\ 1 \end{bmatrix}$ |

**Table 4.1:** The complete PWA representation for the network with 1 layer with 3 nodes. Each region computes its own transformation, with some of the rows zeroed out.

Adding an output layer with no activation is straightforward. The resulting network is shown in figure 4.3. The second layer has no activation, and computes the function:

$$f(x) = \boldsymbol{P}_o \bar{\boldsymbol{x}} = \begin{bmatrix} \boldsymbol{w}_o & b_o \\ 0 & 1 \end{bmatrix} \boldsymbol{P}_i \begin{bmatrix} \boldsymbol{x} \\ 1 \end{bmatrix} = \begin{bmatrix} \boldsymbol{w}_o \boldsymbol{W}_i & \boldsymbol{w}_o b_i + b_o \\ \boldsymbol{0} & 1 \end{bmatrix} \begin{bmatrix} \boldsymbol{x} \\ 1 \end{bmatrix} \tag{4.3}$$

where $P_i$ represents the transformation matrix corresponding to region $i$, as shown in table 4.1. The effect of adding another layer with no activation can be seen as just a multiplication between the parameter matrix of the new layer and the parameter matrices of each region. This suggests that adding layers with no activations will not affect the linear regions. Adding a layer with any number of nodes will have the same form:

$$f(x) = \boldsymbol{P}_o \bar{\boldsymbol{x}} = \begin{bmatrix} \boldsymbol{W}_o & b_o \\ 0 & 1 \end{bmatrix} \boldsymbol{P}_i \begin{bmatrix} \boldsymbol{x} \\ 1 \end{bmatrix} = \begin{bmatrix} \boldsymbol{W}_o \boldsymbol{W}_i & \boldsymbol{W}_o b_i + b_o \\ \boldsymbol{0} & 1 \end{bmatrix} \begin{bmatrix} \boldsymbol{x} \\ 1 \end{bmatrix} \tag{4.4}$$

### 4.1.3 | Example 3: Adding ReLU activation

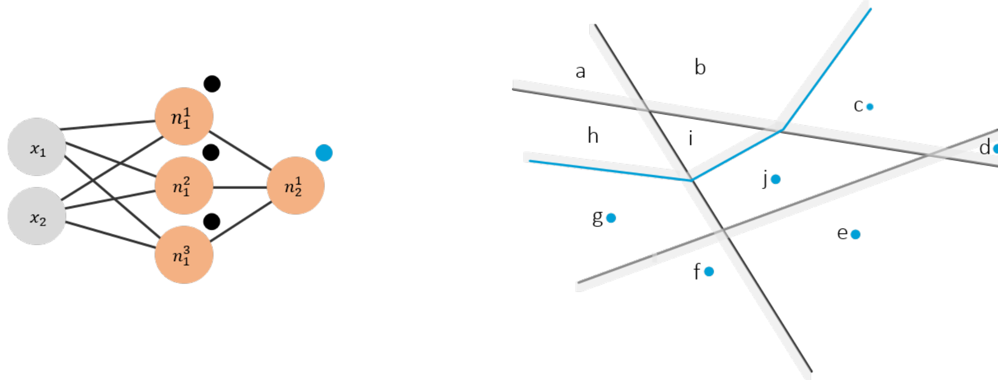



**Figure 4.4:** Simple network with 2 hidden layers. Note how the last node bisects each of its active regions in a different way, yet remains continuous across regions, such that it appears to bend at the boundaries of the previous layer.

An activation is now added to the last node of the network in figure 4.3. The ReLU function deactivates the node in certain regions, switching it on and off. As before, there is a boundary that describes this switching behaviour. However, this time the input space consists of multiple regions defined by the previous layers. Importantly, the parameter matrix $\boldsymbol{P}_i$ for each region is different, implying that the boundary introduced by the last node will be different for each region. The result will be similar to what is depicted in figure 4.4.

The new boundary will be continuous across boundaries (otherwise the PWA function as a whole would be discontinuous!), but it will "bend" as it crosses them. This pattern continues as more layers are added, as new boundaries will bend when intersecting the boundaries of all previous layers. An example is given in figure 4.5.

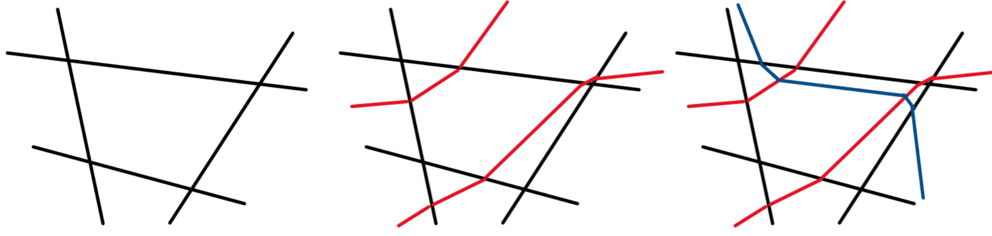


**Figure 4.5:** The hyperplane defined by a node will "bend" at all boundaries of the previous boundaries. The figure shows this effect by adding more layers.

Applying the nonlinearity is relatively simple. Every region found so far is associated with its own unique $\boldsymbol{P}$ matrix, which defines the output of each node in that region. The procedure from the first example can then be applied separately to each region, yielding the new set of regions. The subregions inherit their parents $\boldsymbol{P}$ matrix, with the inactive rows zeroed out.

With this in place, the process of converting the network to its PWA form can be generalised to any number of layers. The missing piece is then the method to find the regions defined by each layer.

### 4.1.4 | Example 4: Finding the regions

The boundaries defined by a layer of nodes can be viewed as a hyperplane arrangement (see section 2.3.4). The regions of the arrangement can be expressed as the set difference $\mathbb{R}^d \backslash H$, where $H$ is the union of all the hyperplanes. However, the MPT toolbox does not yield the correct partition when performing this operation. It was found that this is because it does not support the representation of open sets.

One solution is to approximate the hyperplanes using thin, full-dimensional regions. Such a region can be constructed as the space between the hyperplane and a very slightly shifted copy. If the hyperplane has the form $\boldsymbol{W}\boldsymbol{x} = \boldsymbol{b}$, then the thin region can be expressed as:

$$\{x \mid \boldsymbol{W}\boldsymbol{x} \leq \boldsymbol{b}\} \cap \{x \mid -\boldsymbol{W}\boldsymbol{x} \leq -\boldsymbol{b} + \epsilon\}, \quad 0 < \epsilon << 1 \tag{4.5}$$

It was found that $\epsilon = 10^{-7}$ worked well. Values lower than $\epsilon = 10^{-8}$ fell below the tolerance level used by the MPT toolbox, and were ignored.

While this approach is simple, it resulted in numerical errors later due to the gaps it introduces between regions. A more exact partitioning algorithm was needed.

As explained in section 2.3.4, finding the intersection of two convex polyhedra in the H-representation is cheap, as it only involves the concatenation of their constraints. Thus, given a polyhedron $P$ and a hyperplane $\boldsymbol{W}\boldsymbol{x} = \boldsymbol{b}$ that bisects $P$, it is simple to find the 2 subregions.

$$\begin{aligned} \Omega^+ &= P \cap \{\boldsymbol{x} \mid \boldsymbol{W}\boldsymbol{x} \leq \boldsymbol{b}\} \\ \Omega^- &= P \cap \{\boldsymbol{x} \mid -\boldsymbol{W}\boldsymbol{x} \leq -\boldsymbol{b} \end{aligned} \tag{4.6}$$

The question is thus reduced to whether or not the hyperplane intersects the polyhedron. Fortunately, this can also be obtained cheaply. Replacing $\boldsymbol{W}\boldsymbol{x} = \boldsymbol{b}$ with 2 equivalent inequalities (see section 2.3.4), the H-representation of the intersection is obtained by concatenating all of the constraints. If there is no feasible point, then there is no intersection, which can be verified using a feasibility LP (see appendix A).

The regions of a hyperplane arrangement can therefore be found in an iterative fashion, as shown in figure 4.6. For every hyperplane, iterate through the previously found regions and check for intersections. If there is an

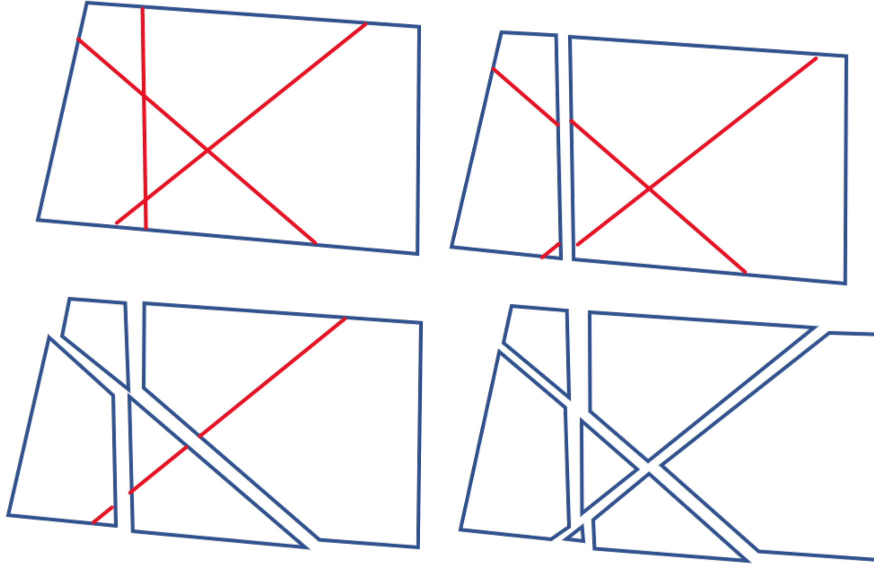


**Figure 4.6:** Example of how the regions of a hyperplane arrangement can be found iteratively. While simple, the drawback is that the number of bisections performed will increase drastically with the number of hyperplanes.

intersection, find the two subregions, and insert them into the list of regions, removing the old region. This algorithm forms the core of how the ReLU nonlinearity is applied in the full algorithm. A formal write-up can be seen in figure 2.

Using this procedure, the number of observed regions was measured for a variety of different input dimensions and arrangements. The results can be seen in 4.7. The runtimes of the algorithm is omitted here, as this is investigated more fully in section 4.4.

## 4.2 | Algorithm

As seen in section 4.1, a neural network can be converted to its PWA representation in an iterative fashion, starting at the input layer. Each subsequent layer is then parsed, building up a collection of known linear regions. The algorithm can be summarised as:

1. Get the next layer
2. Multiply the $\boldsymbol{P}$ matrices of all currently known regions with the $\boldsymbol{P}$ matrix of the layer
3. If the layer has an activation (ReLU), iterate over all known regions and partition them using the procedure from Example 1.
4. The $\boldsymbol{P}$ matrices of all new regions are updated by finding interior points and checking which rows of $\boldsymbol{P}$ are inactive. The inactive rows are set to zero.
5. If there are more layers, go to point 1. Otherwise, return the set of known regions.

The set of currently known regions and their corresponding $P$ matrices is called the **working set** $\mathcal{W}$. Every element in $\mathcal{W}$ will be a tuple of the form $(D, \boldsymbol{P})$, where $D$ is the domain and $\boldsymbol{P}$ is the parameter matrix that defines the affine transformation computed within that region.

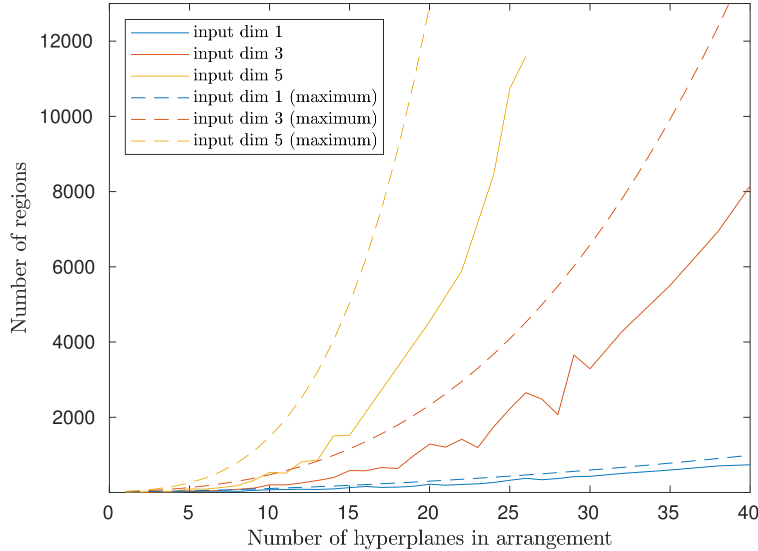



**Figure 4.7:** Measured average number of regions vs the theorectical maximum according to Zavslavsky's Theorem (see (2.2)). The average number of regions diverges from the maximum significantly as the number of hyperplanes increases. Increasing the dimension also appears to increase this gap.

The neural network $\mathcal{N}$ is also represented as a list of tuples of the form $(\boldsymbol{P}^N, \sigma)$, which represent the layers. $\boldsymbol{P}^N$ is the transformation performed by the connection weights, and $\sigma$ is the activation function (if any). $\sigma$ is always assumed to be the ReLU function.

Collections (discrete sets) of parameter matrices $\boldsymbol{P}_i$ and regions $D_i$ are labeled $\mathcal{P}$ and $\mathcal{D}$ respectively. To make the full algorithm more readable, it has been split into three operations.

(i) **layer_compose()**: When encountering a new layer, this operation applies the corresponding affine transformation to all regions in the working set $\mathcal{W}$.

(ii) **repartition()**: Apply the nonlinearity by computing the boundaries $\boldsymbol{w}_i^T \boldsymbol{x} = -b_i$ and solving the hyperplane arrangement problem, as described in 4.1.4.

(iii) **calculate_P()**: Once the new regions have been found, the new parameter matrix for each of them is calculated by zeroing out the inactive rows of the previous parameter matrix.

An overview of the algorithm is shown in Algorithm 1. The operations are given in Algorithm 2. The notation follows the conventions described in the preliminaries (section 2.2).

The main loop iterates over the layers of the network, performing all three operations exactly once for each of the regions currently in the working set (if the layer has an activation, typically true for all layers, excepting the last). As the size of the working set will increase after processing each layer, it is clear that the worst case performance of the algorithm will depend greatly on the total depth of the network. However, it is not clear how much the working set will increase. For example, some regions in the working set may be intersected multiple times by the node boundaries in the next layer, while others will not be intersected at all. The runtimes are investigated further in 4.4.

1. **layer_compose()**: Thanks to the compact notation from 2.3.6, this operation involves one matrix multiplication per region in the working set $\mathcal{W}$. The cost will increase linearly with the size of the working set, but it will be far lower than the other operations. For example, performing 10000 multiplications with $[100 \times 100]$ matrices takes around 0.5 seconds on a laptop.

2. **calculate_P()** is also linear wrt to the regions in $\mathcal{W}$, and performs one matrix multiplication. How-

**Algorithm 1:** Overview of the PWA conversion algorithm.

**Result:** $\mathcal{W} = \big((D_1, \boldsymbol{P}_1), \ldots, (D_n, \boldsymbol{P}_n)\big)$
**begin** definitions
  $\mathcal{N} = \big((\boldsymbol{P}_1^N, \sigma_1), ..., (\boldsymbol{P}_n^N, \sigma_n)\big)$ = layers of neural network
  $d$ = input dimension
**end**
**fn** `main`($\mathcal{N}$) **is**
  $\mathcal{W} \leftarrow (\mathbb{R}^d, I_{d+1})$
  **foreach** $(\boldsymbol{P}_k^N,\ \sigma_k) \in \mathcal{N}$ **do**
    $\mathcal{W} \leftarrow$ ( `layer_compose`($\mathcal{W}$, $\boldsymbol{P}_k^N$) )
    **if** $\sigma$ **then**
      $\mathcal{W}_{new} \leftarrow \emptyset$
      **for** $(D_i,\ \boldsymbol{P}_i) \in \mathcal{W}$ **do**
        $\mathcal{D}_{new} \leftarrow$ `repartition`($D_i$, $\boldsymbol{P}_i$)
        $\mathcal{P}_{new} \leftarrow$ `calculate_P`($\mathcal{D}_{new}$, $\boldsymbol{P}_i$)
        $\mathcal{W}_{new} \leftarrow \mathcal{W}_{new} \parallel$ `zip`($\mathcal{D}_{new}$, $\mathcal{P}_{new}$)
      **end**
      $\mathcal{W} \leftarrow \mathcal{W}_{new}$
  **end**
**end**

ever, it must obtain an interior point of each region. The interior point for a polyhedron with an H-representation is calculated by solving an LP to find the Chebyshev centre of the region (see 2.3.4). This can be solved efficiently. As an example, for 1000 non-empty 12-dimensional regions with 50 constraints each and a random parameter matrix with 500 rows, this operation takes around 0.01 seconds on a laptop.

3. **repartition()** solves the hyperplane arrangement problem for some initial region by iteratively bisecting it into subregions with a hyperplane. The main source of complexity is that it must check for intersections with all of the subregions that it has identified so far, the number of which quickly explodes, as shown in equation (2.2). Section 4.4 investigates the runtime of **repartition()** in some detail.

## 4.3 | Optimisations

In order to improve the performance of the algorithm, two optimisations were made. The first optimisation improved **repartition()** by reducing the number of intersection checks. The second optimisation involved parallelising the algorithm, which was extremely effective.

(i) **Reducing intersections check by repartition()**: The original algorithm described in 4.1.4 considers each hyperplane alone, checking if it intersects all currently known regions. When it finds an intersection, it divides the region and replaces it with the subregions. The optimised code instead organises the regions in a tree structure, and storing the subregions as children of a parent region. If a hyperplane does not intersect a parent region, then it will not intersect any of the subregions. In this way, the number of intersection checks can be reduced significantly. The result is reconstructed at the end by performing a depth first search of the tree, returning the leaf nodes. The improved algorithm is vastly more efficient for larger hyperplane arrangements. However, as will be shown in section 4.4, this did not reflect in the runtimes of the main algorithm.

(ii) **Parallelisation**: It was noted that the algorithm could be easily parallelised when the working set

**Algorithm 2:** Elementary operations required for the PWA conversion

```
fn layer_compose(𝒲, T) is
    foreach (𝒟_i, P_i) ∈ 𝒲 do
        P_i ← T P_i
    end
    return 𝒲
end
fn repartition(D, P) is
    𝒟 ← {D}
    foreach row (w_j, b_j) ∈ P do
        B ← {x | w_j^T x + b_j = 0}
        D_active ← {x | w_j^T x ≥ −b_j}
        D_inactive ← {x | w_j^T x < −b_j}
        𝒟_new ← 𝒟
        foreach D_i ∈ 𝒟 do
            /* If the boundary intersects this region...   */
            if D_i ∩ B ≠ ∅ then
            𝒟_new ← 𝒟_new ‖ (D_i ∩ D_active) ‖ (D_i ∩ D_inactive)
        end
        𝒟 ← 𝒟_new
    end
    return 𝒟
end
fn calculate_P(𝒟, P) is
    foreach D_i ∈ 𝒟 do
        x ← interior_point(D_i)
        y ← P x
        P_new ← P
        foreach y_i ∈ y do
            if y_i ≤ 0 then
                /* Set the ith row of P_new to zero   */
                P_new[i, :] ← 0
        end
    end
end
```

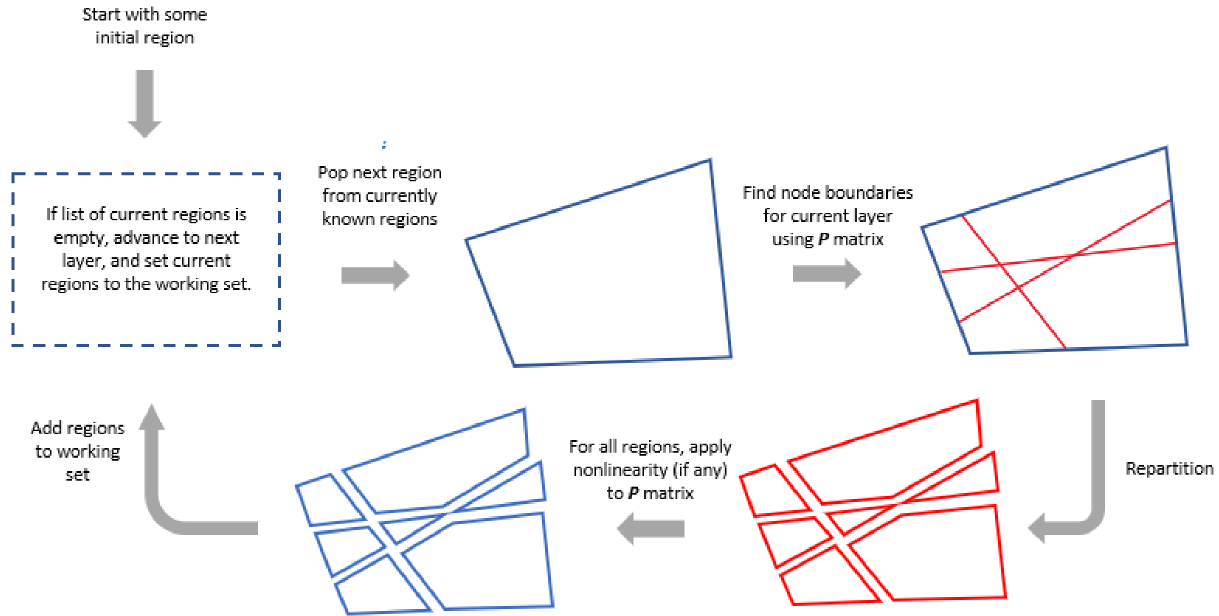



**Figure 4.8:** Overview of the algorithm. The node boundaries of each layer form a hyperplane arrangement in all of the regions of the previous layers.

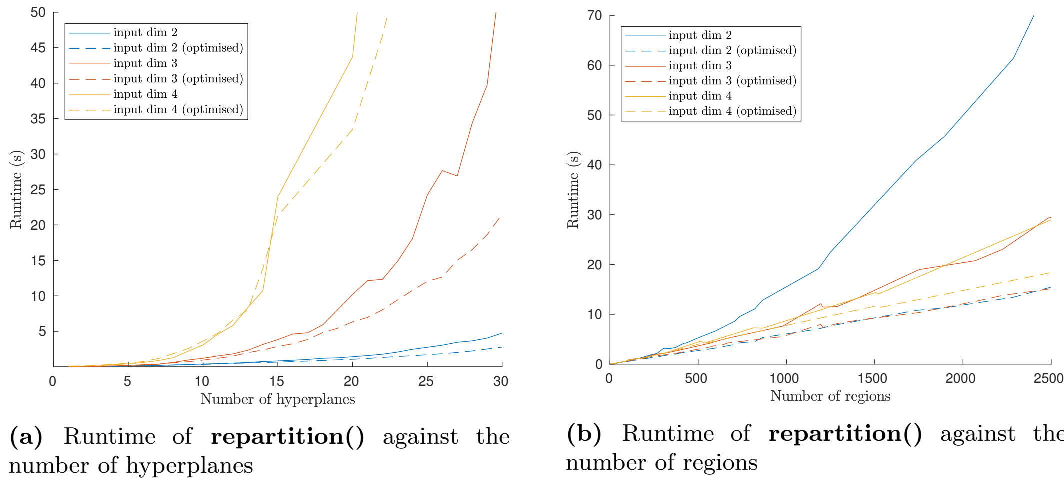
contained more than one region. Each region in the working set has its own parameter matrix which is independent from the rest of the regions. This allows **repartition()** to be run on each region independently. The results can safely be concatenated afterwards, as there is no chance of the subregions overlapping. This optimisation was particularly effective.

## 4.4 | Runtime

All runtimes were measured using a machine with a 6-core, 3,5 GHz processor and 16 GB of RAM. The results for **repartition()** and its optimised version in terms of the number of hyperplanes have been presented together in figure 4.9a. The runtimes are also presented in terms of the number of regions in figure 4.9b. This shows that the runtime is roughly proportional to the number of regions found.

The optimisation of **repartition()** focused on reducing the number of intersection checks that it performs. These checks were counted, and the results are shown in figure 4.11. The relative improvement of the optimisation is shown in figure 4.10.

The runtime of the main algorithm was measured with/without the optimisations and with/without parallelisation. Networks up to input dimension 4 were recorded, as the number of regions quickly exploded and the runtimes became too long.




**(a)** Runtime of **repartition()** against the number of hyperplanes

**(b)** Runtime of **repartition()** against the number of regions

**Figure 4.9:** The runtime of **repartition()** increases significantly with the number of nodes/hyperplanes. The optimisation yields significant savings for higher numbers of hyperplanes. The runtime also appears to be somewhat linear wrt to the number of regions found, as expected.

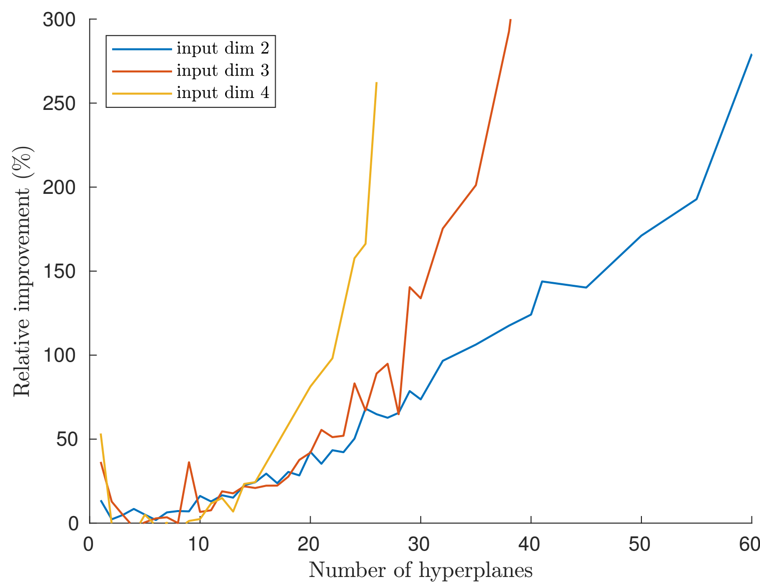



**Figure 4.10:** Relative improvement in runtime after optimisation. It appears to scale extremely well with the number of regions, and the benefits are more pronounced with larger input dimensions.

Figure 4.9b shows that the runtime complexity is directly proportional to the number of regions found. A more surprising result is that the effect of increasing the dimension (and thus the size of the required LPs) is almost negligible in comparison. This suggests that it is the high number of calls to the LP solver, rather than the size of the LPs, that dominates the time complexity.

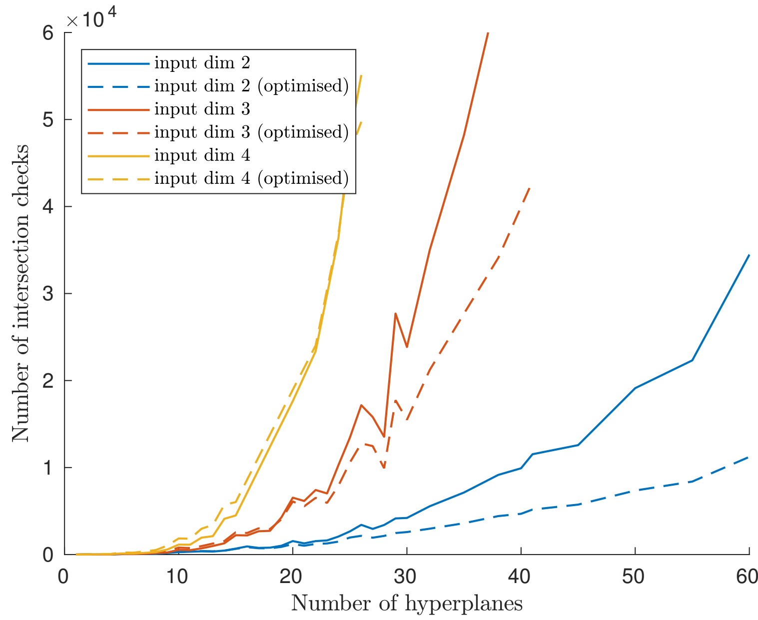
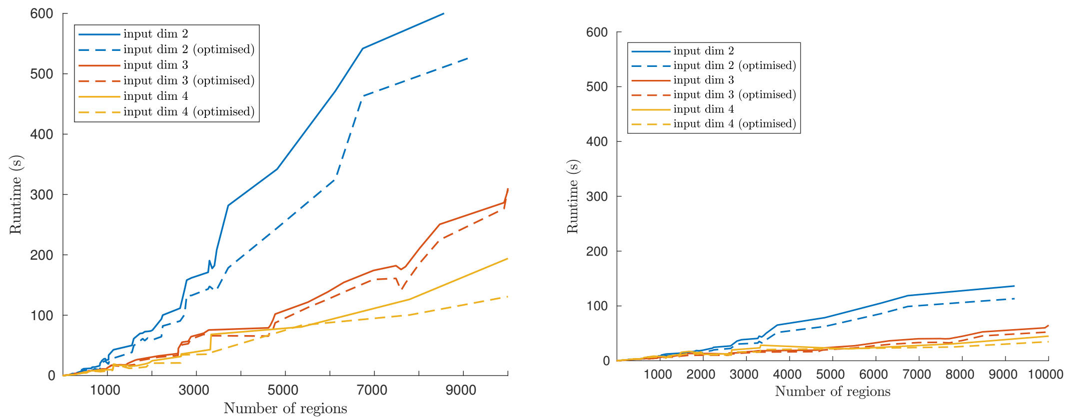



**Figure 4.11:** Number of intersection checks performed before and after the optimisation. As seen before, the optimisation is more effective for larger problems.




**(a)** Runtime of **main()** against the number of regions found

**(b)** Runtime of **main()** after parallelisation on a 6-core machine

**Figure 4.12:** Despite the large performance gains the optimisation of **repartition** promised, it only gives a slight performance boost in the main algorithm. This is likely due to the small size of the subproblems that are passed down to **repartition()**. Parallelisation was much more effective, with the performance increasing by a factor almost equal to the number of cores used.

# 5 | Case Study Results

## 5.1 | Case study I: Simple LTI system

As a proof of concept, this study aims to control some simple system using a small neural network, obtain the PWA form of the closed loop system, and then to apply the Lyapunov method to verify stability. Consider the following LTI:

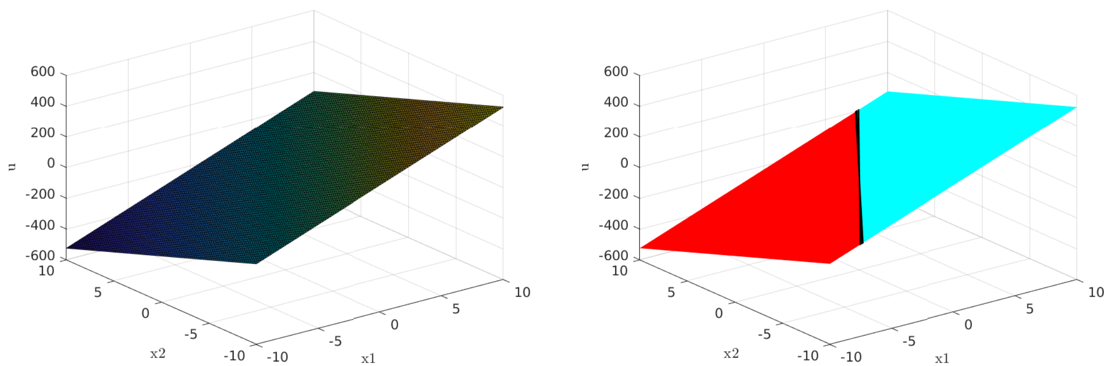
$$\dot{\boldsymbol{x}} = \begin{bmatrix} 2 & -0.1 \\ 0.2 & 3 \end{bmatrix} \boldsymbol{x} + \begin{bmatrix} 0.1 \\ 0.2 \end{bmatrix} u \tag{5.1}$$

An optimal control law was computed using LQR (see section 2.3.3 using $Q = \boldsymbol{I}_2$ and $R = 1$. The resulting linear control law was found to be:

$$u = -\boldsymbol{K}\boldsymbol{x} = -\begin{bmatrix} -29.98 & 22.41 \end{bmatrix} \boldsymbol{x} \tag{5.2}$$

The control law $u(x)$ and the network output can be plotted as 3D surfaces, shown in 5.1a. The closed loop system is $\dot{\boldsymbol{x}} = (\boldsymbol{A} - \boldsymbol{B}\boldsymbol{K})\boldsymbol{x}$, and the eigenvalues of $(\boldsymbol{A} - \boldsymbol{B}\boldsymbol{K})$ are $-0.2207$ and $-0.7622$. The system is GAS because the eigenvalues are both negative.

A small neural network was trained to imitate this control law. The network had 1 hidden layer with 5 nodes and ReLU activation. The training data was generated by randomly sampling $u(x)$ 50000 times using the normal distribution $\mathcal{N}(0, 100)$. The SGDM optimiser was used with a learning rate of 0.001. After 30 epochs, the loss was $2.1565 \cdot 10^{-8}$. The network was then converted to its PWA representation (see figure 5.1b). Qualitatively, the PWA representation of the network appears to be identical to $u(x)$.



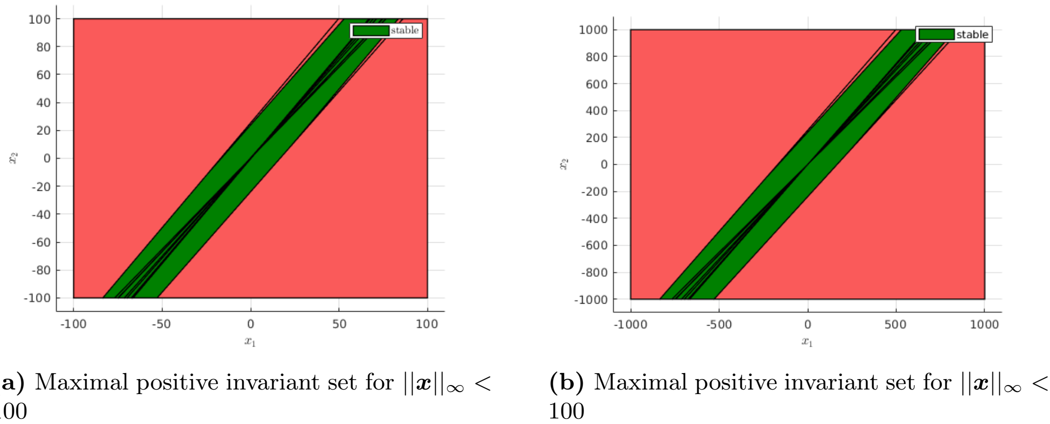

**(a)** Optimal LQR control law $u(x) = -\boldsymbol{K}\boldsymbol{x}$ **(b)** PWA representation of imitation neural network

**Figure 5.1:** All hyperplanes have congregated together in the middle of PWA representation of the neural network, producing a flat output.

With the PWA representation of the network in hand, the closed loop dynamics with respect to 5.1 were found. An attempt was made to find a Lyapunov function that could prove global stability. However, this proved to be an infeasible problem, as the full closed loop dynamics did not lie within a positive invariant set. To solve this, the maximum positive invariant set was found using the MPT toolbox. However, due to software limitations this can only be done for finite sets, which requires that additional constraints be introduced. The simplest approach is to just restrict the state to lie within some box.

Two different sets of constraints were chosen, $||\boldsymbol{x}||_\infty < 100$ and $||\boldsymbol{x}||_\infty < 1000$, and the maximum positive invariant set was computed for both. The results are shown in figure 5.2. Interestingly, the invariant set appeared scale-invariant, as relaxing the constraints had no effect on the appearance of the invariant set. This suggested that as $\boldsymbol{x} \to \infty$, the size of the invariant set might grow indefinitely, eventually covering all of $\mathbb{R}^2$. This could not be verified using the selected tools. Despite this, the positive invariant set can be made as large as needed, and a stability verification can be performed for the selection.

With the constraints in plane, another attempt to find a Lyapunov function was made, and a feasible solution was found (see figure 5.3a, thereby showing asymptotic stability within the positive invariant set.



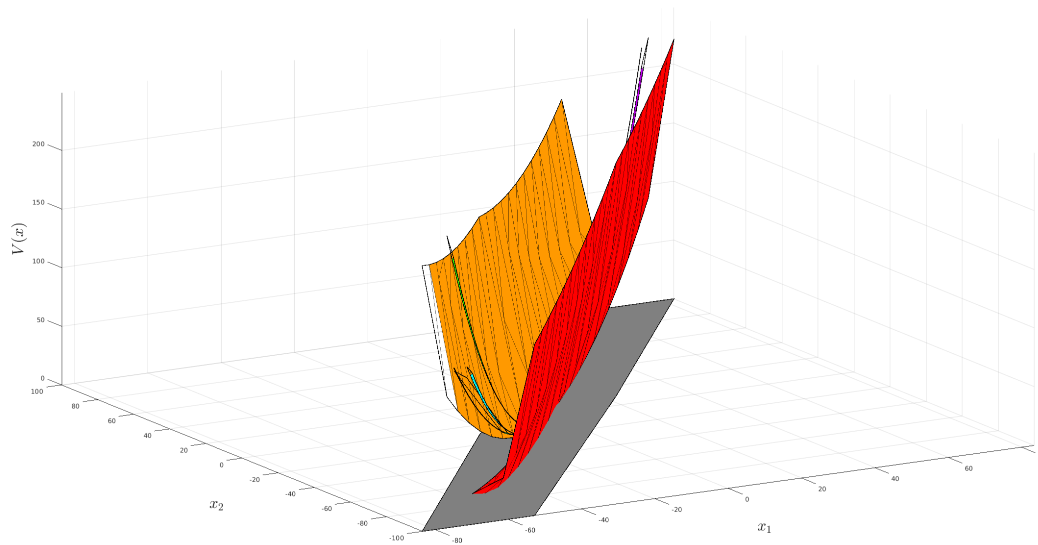

**(a)** Maximal positive invariant set for $||\boldsymbol{x}||_\infty < 100$

**(b)** Maximal positive invariant set for $||\boldsymbol{x}||_\infty < 100$

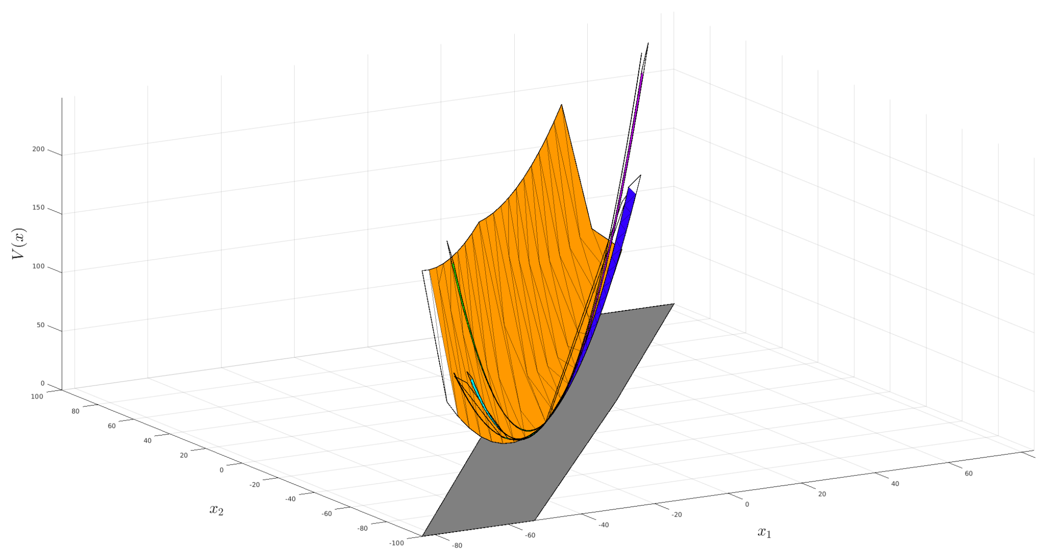
**Figure 5.2:** Maximum positive invariant set computed for two different state constraints. The set appears scale-invariant, suggesting that the stable region might grow indefinitely with state constraints. Note that additional boundaries are introduced by the process of finding the invariant set; these correspond to the transition regions within each region (see 2.3.5).

### Comments

- Despite the simplicity of the system, the stability analysis was involved than expected. First, a finite subset of the PWA regions had to be specified by placing constraints on the state, then a positive invariant set had to be computed, before searching for a Lyapunov function. The benefit of this approach is that it can be applied to any autonomous PWA system.
- It was also shown that global stability is difficult to prove, as it is only practical to work with finite positive invariant sets. This is only a theoretical issue however, as real-world systems will have state constraints. If a positive invariant subset within the constraints can be found, then stability can be verified.




**(a)** Lyapunov function for the LTI system




**(b)** Lyapunov function with one of its pieces removed

**Figure 5.3:** The function is given a valley shape defined by the two largest regions, while the internal regions are thin slivers. The same Lyapunov function, but with one of the valley "walls" removed so that the internal pieces can be seen better.

## 5.2 | Case study II: Hybrid model for Unforced Pendulum

This case study investigates whether the procedure used in Case I may be applied to larger, more complex systems. To this end, a neural network was used to model the dynamics of an autonomous nonlinear system. The model was chosen to be a simple pendulum:

$$\ddot{\theta} = -\frac{g}{L}\sin\theta - \frac{d}{m}\dot{\theta} \tag{5.3}$$

Where $g = 9.81$, $m = 1$, $L = 5$, and $d = 0.1$. Letting $x_1 = \theta$ and $x_2 = \dot{\theta}$ the following system of first order ODEs is obtained:

$$\dot{\boldsymbol{x}} = \begin{bmatrix}\dot{x}_1 \\ \dot{x}_2\end{bmatrix} = \begin{bmatrix} x_2 \\ -\frac{g}{L}\sin x_1 - \frac{d}{m}x_2\end{bmatrix} \tag{5.4}$$

Suppose now that only measurement data of the above is available, and that it is for some reason difficult/-expensive to develop an analytical model for the system. A function approximator like a neural network can then be fit to the data, and analysed instead.

To simulate this scenario, a similar procedure to Case I was followed. $\theta$ and $\dot{\theta}$ were randomly sampled from the continuous uniform distribution $\mathcal{U}(-\pi, \pi)$ and the normal distribution $\mathcal{N}(0, 5)$ respectively. Equation 5.4 was then used to generate a dataset with 50000 data samples.

A neural network with 2 hidden layers (15 and 10 nodes respectively) was then trained on the data using the ADAM optimiser with a learning rate of 0.003 for 50 epochs. The trained model achieved a final loss of $1.615 \cdot 10^{-4}$ on the training data. A comparison between the learned and real dynamics via can be seen in figure 5.4.

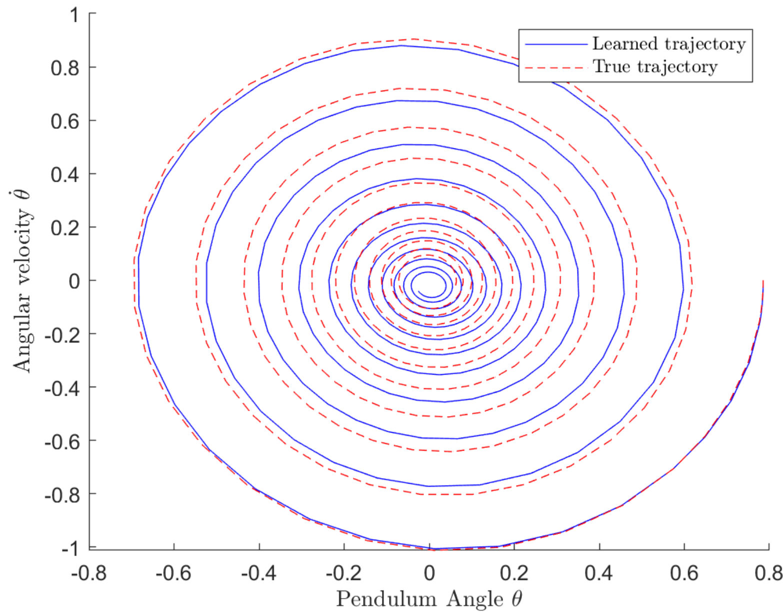



**Figure 5.4:** True and simulated trajectory using the neural network with $\theta_0 = \frac{\pi}{4}$. The network displays some asymmetries in its trajectory, suggesting that the learned pendulum would swing slightly higher on one side. It also appears to converge slightly off-centre of the origin. This is due to the fact that the neural network does not assume energy conservation.

The neural network was then converted to its PWA representation using the conversion algorithm, the result of which is shown in figure 5.6a. The representation had 116 linear regions in total. The maximal positive invariant set was then calculated using the MPT toolbox. An overview of this procedure can be found in the appendix B. To ensure convergence of the computation, the following state constraints were applied[1].

$$\begin{aligned} \theta &\in [-2.9\pi, 2.9\pi] \\ \dot{\theta} &\in [-10, 10] \end{aligned} \tag{5.5}$$

The maximal positive invariant set for these constraints can be seen in figure 5.5. Next, an attempt was made to find a Lyapunov candidate. However, this turned out to be infeasible. This was unexpected, as there is a well defined invariant set and figure 5.4 seems to suggest that the simulated system is asymptotically stable. To investigate this, the outputs of the network were inspected. The two outputs are plotted separately in figures 5.6b and 5.6c. The first output seems to match equation (5.4), simply outputting $\dot{\theta}$. The second output also appears to match the model. However, on closer inspection the output is quite irregular, as shown in figure 5.7.

As mentioned in 2.3.5, a Lyapunov candidate may be piecewise discontinuous, as long as it always decreases when transitioning from one region to another. The irregularity in figure 5.7 likely makes it difficult or impossible to satisfy this decrease condition.

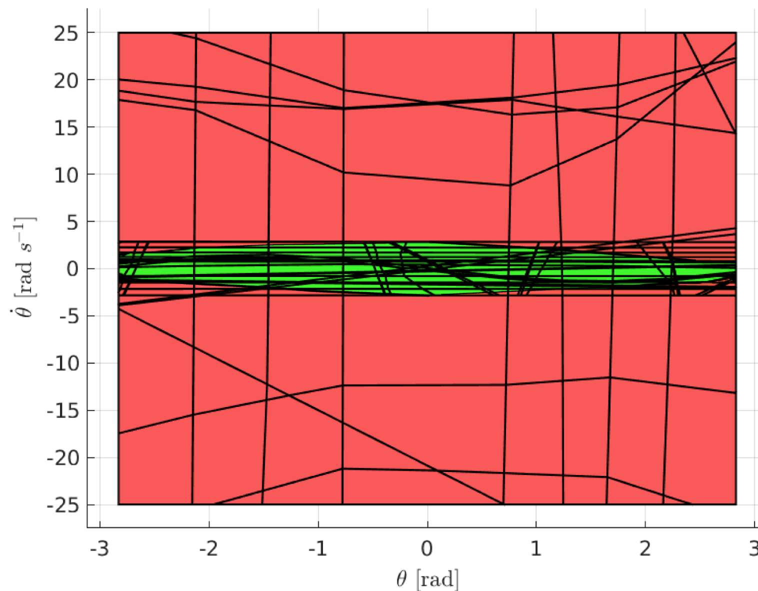
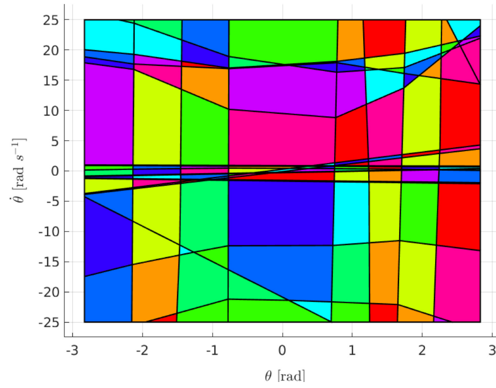
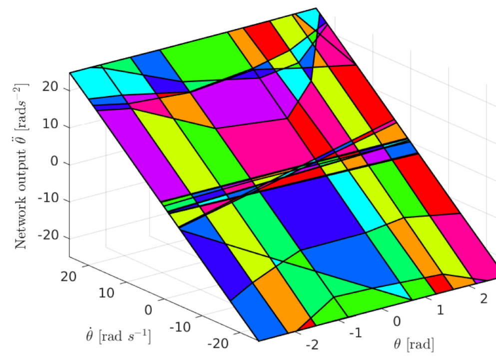



**Figure 5.5:** Invariant regions of the pendulum model. Many additional boundaries were created when mapping the transitions between regions. Note that model was only trained on the interval $\theta \in [-pi, pi]$. In order to make the invariant set calculation more reliable, the interval was further restricted to $\theta \in [-2.9, 2.9]$.. If the angular velocity of the pendulum is too high, then it will exit the known regions, which corresponds to the pendulum doing a full rotation. This takes the state outside of the "known regions", such that regions near $\theta = \pm\pi$ become invariant.

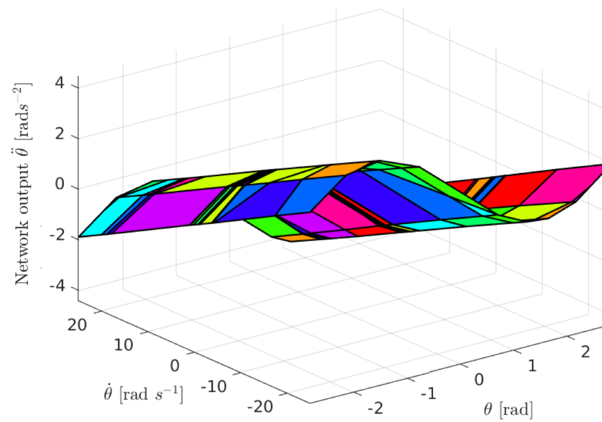
[1] The interval for $\theta$ was chosen to be $[-2.9, 2.9]$ in order to avoid numerical difficulties at $\theta = \pm\pi$.




**(a)** Linear regions (116 total) of the pendulum neural network




**(b)** First output of the network: $\dot{\theta}$




**(c)** Second output of the network: $\ddot{\theta}$

**Figure 5.6:** The most distinctive features of (a) are the large vertical strips and the concentration of boundaries along the x-axis. The outputs of the network help explain this structure. The large vertical strips form sheets that create the sinusoidal shape of $\ddot{\theta}$, while the horizontal boundaries collected in the centre contribute to the flat slope of $\dot{\theta}$.

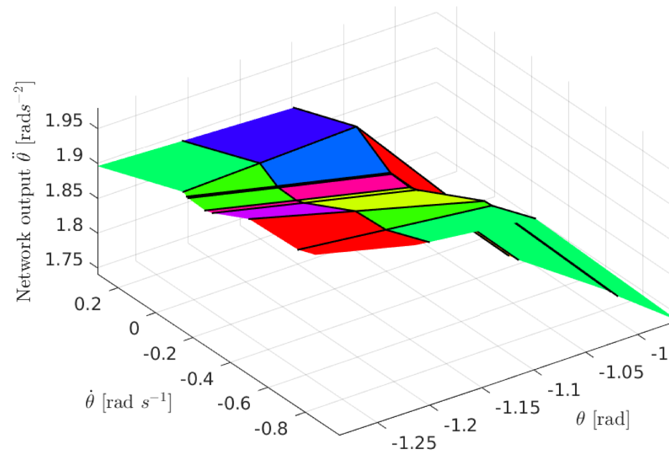



**Figure 5.7:** Close up of the second output of the network. While the output appears to match the model on the large scale, on the smaller scale it appears quite irregular. This is likely to be the reason why finding a Lyapunov candidate was infeasible.

### Comments

This particular study highlights at least two major pitfalls when using neural networks in a modeling context:

- A pure data driven modeling approach based on conventional neural networks does not take into account physical laws. It instead assumes that these laws are implicitly captured in the data used for training which might not always be the case[2].
- The output of neural networks can be quite irregular at small scales, especially in regions that seem to have a high concentration of small linear regions. This makes more complex neural network models difficult to analyse using standard techniques such as the Lyapunov method.

[2] A raw dataset may contain biases that can make it inherently unphysical

## 5.3 | Case Study III: Applying MPC to a data-driven pendulum model

The previous studies demonstrated that applying standard stability verification methods to autonomous, neural network models is challenging, even when they can be explicitly expressed as a PWA function. This case study instead considers the use of MPC with forced, neural network models. This also brings to light the differences between two formulations of PWA systems described in 2.3.5, and the importance of choosing the correct one.

Shown below is a forced model of the pendulum from case II, with the coordinates shifted such that $\boldsymbol{x}$ corresponds to the pendulum being at rest in the upright position.

$$\dot{\boldsymbol{x}} = \begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} x_2 \\ -\frac{g}{L}\sin\pi + x_1 - \frac{d}{m}x_2 + \frac{1}{m}u \end{bmatrix} \tag{5.6}$$

The parameters are the same as in Case II, where $g = 9.81$, $m = 1$, $L = 5$, and $d = 0.1$. One difference is that $\theta = 0$ now corresponds to the upright position. This was done to simplify the MPC formulation (see 2.3.3).

Two different data-driven models were created. The first was obtained by training a neural network using the same procedure and network architecture from Case II, but with the coordinate shift. The network was then converted to its PWA form and augmented with an input term. The second model was created by training a neural network directly on the forced dynamics, and then converting it to the PWA representation.

The data for the simpler model was generated by sampling $\theta$ and $\dot{\theta}$ from the continuous uniform distribution $\mathcal{U}(-2\pi, 2\pi)$ and the normal distribution $\mathcal{N}(0, 5)$ respectively, similarly to what was done in Case II. The output data was generated using (5.6), setting $u = 0$. The neural network was given 2 hidden layers (15 and 10 nodes respectively) and trained on the data using the ADAM optimiser with a learning rate of 0.003 for 50 epochs. The PWA form of the model was then obtained using teh PWA conversion algorithm:

$$\dot{\boldsymbol{x}} = \boldsymbol{A}_i\boldsymbol{x} + f_i = \begin{bmatrix} a_1 & a_2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + f_i, \quad \boldsymbol{x} \in \Omega_i \tag{5.7}$$

The matrices $\boldsymbol{A}_i$ can be understood as the gradient of the model in each linear region $\Omega_i$, while $f_i$ is the offset from the origin. The model (5.7) was then augmented by adding the $\frac{u}{m}$ term to the affine transformation computed in each region:

$$\dot{\boldsymbol{x}} = \bar{\boldsymbol{A}}_i\boldsymbol{x} + f_i = \begin{bmatrix} a_1 & a_2 & \frac{1}{m} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ u \end{bmatrix} + f_i, \quad (\boldsymbol{x}, u) \in \Omega_i \tag{5.8}$$

Note that this involves augmenting the state of the system to include $u$. Therefore, the regions themselves had to be modified as well, in order to accommodate the additional variable. This involved lifting the dimension of each region. For example, lifting a flat square to the $3^{rd}$ dimension would stretch it into an infinitely long cuboid. This was done by adding a zero column to the H-representation of each region in the $u$ position, which is equivalent to adding $u$ as a free variable to the linear inequalities that define each region. The resulting linear regions are shown in 5.8. Unfortunately, the output of the network could not be plotted as a surface, as was done in Case II.

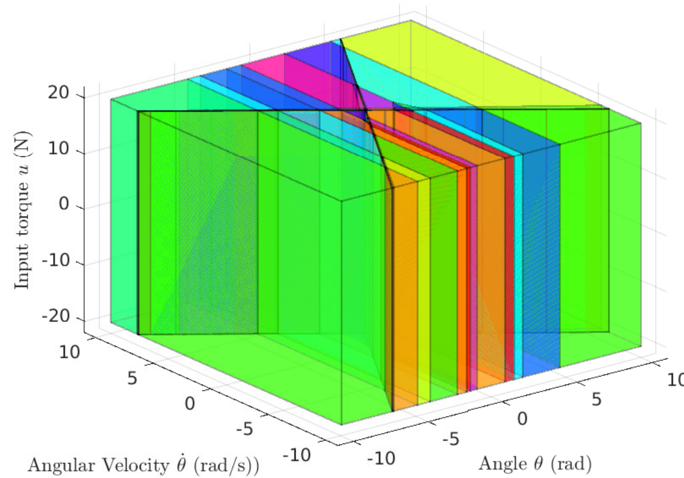



**Figure 5.8:** The linear regions of the simple model are shown here with a slight transparency. As the neural network only has 2 inputs, its linear regions are 2-dimensional. An input term was included in the model, and the regions from were thus lifted a dimension. Note that changing $u$ alone (moving vertically in this case) will never cause the system to transition between regions. Also note that the regions stretch out to infinity in the vertical direction, but have been clipped for visibility. The partition has vertical strips similar to the ones seen in Case II, figure 5.6a. However, while the Case II network had a concentration of horizontal boundaries, this network seems to have created an 'X' shape instead.

The dataset for the more complex model was created by randomly sampling $\theta$, $\dot{\theta}$, and $u$, and using (5.6) to generate the corresponding outputs. $\theta$ and $\dot{\theta}$ were randomly sampled from the continuous uniform distribution $\mathcal{U}(-2\pi, 2\pi)$ and the normal distribution $\mathcal{N}(5)$ (same as Case II), while $u$ was sampled from the normal distribution $\mathcal{N}(0, 5)$. 50000 data samples were generated. Several networks were trained, in an attempt to minimise the network size. The final network was given 3 hidden layers with ReLU activation, with 20, 15, and 10 nodes respectively. All networks were trained on the dataset for 50 epochs using the ADAM optimiser with a learning rate of 0.009. The resulting 1182 linear regions can be seen in figure 5.9.

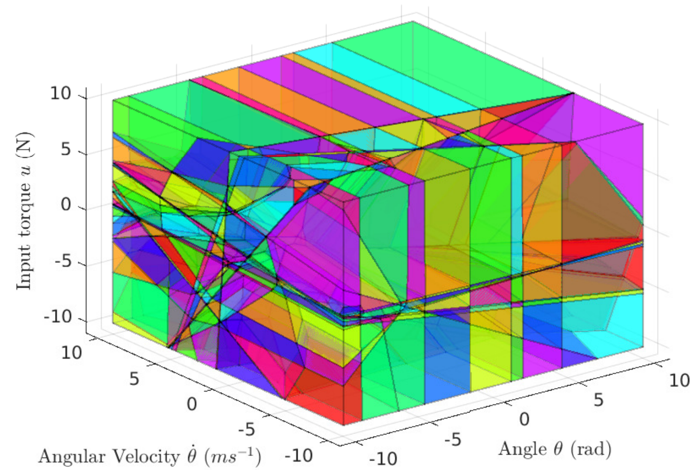
The two systems were simulated with no input, as can be seen in figure 5.10. Despite having over 10 times the regions, the larger network does not achieve significantly better simulation results. Additionally, while some of the boundaries have arranged themselves to form similar vertical strips to the ones found in Case II and the simple network, the rest of the boundaries exhibit little discernible structure.

An MPC controller was attached to both PWA systems. With an initial condition of $\begin{bmatrix}\theta & \dot{\theta} & u\end{bmatrix} = \mathbf{0}$, the control objective was set to $\theta = 0$, which involves rotating the pendulum to the upright position and keeping it there. For a horizon $N = 10$, the simpler augmented model took around 100 seconds on average to complete a single iteration of MPC. In contrast, the larger model took around 30 minutes with a prediction horizon of $N = 2$! Because of this, the closed loop simulations were deemed impractical.

### Comments

This case study shows that:

- Although neural networks have the ability to learn complicated dynamics from data alone, they tend to partition the state space in an unstructured, complicated fashion.
- The complexity arising from the excessive partitioning can be mitigated by reducing the dimension of the network and adding additional terms directly to the PWA representation. This can also be seen as




.

**Figure 5.9:** The linear regions of the network with 3 dimensions are shown here with some slight transparency. The partition is immensely complex, with 1182 regions, an increase by a factor of 10 compared to Case II! A change in state any direction is likely to cause a mode transition. The structure is not simple to explain, as it was for case II (see figure 5.6c)

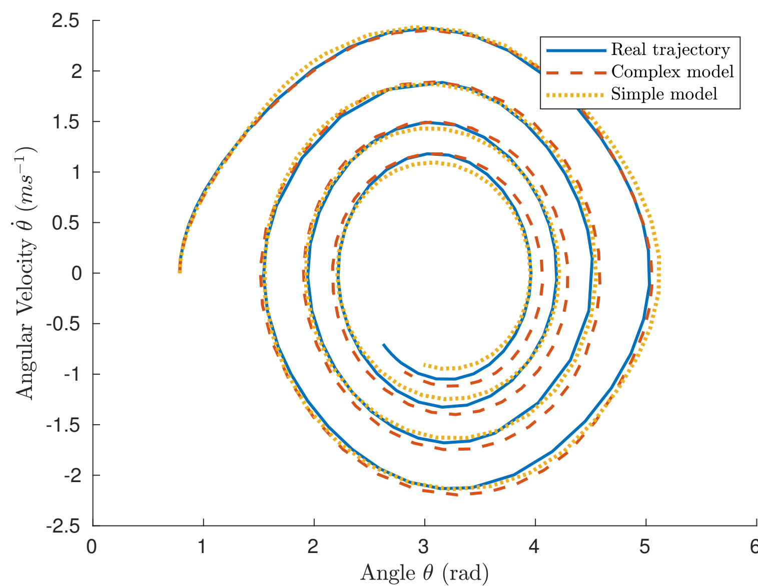



**Figure 5.10:** Both the simple and the complex model were simulated alongside the actual model, with the initial state $\boldsymbol{x} = (\frac{\pi}{4}, 0)$. Note that $\theta = 0$ corresponds to the upright position, while $\theta = \pm\pi$ is the stable bottom position. Both the simple model and the complex model have similar modeling capabilities, despite the differences in complexity.

a hybrid modeling approach where domain knowledge / governing equations are used in combination with a data-driven components.

- The PWA conversion algorithm can be used to express the data-driven components of these relatively simple hybrid models as PWA functions. If the rest of the hybrid model is PWA or linear, then the whole model has an explicit PWA form.
- These PWA systems are still challenging to work with, as demonstrated by the attempt to apply MPC on these models.

## 5.4 | Case Study IV: Closing the loop using neural network models and controllers

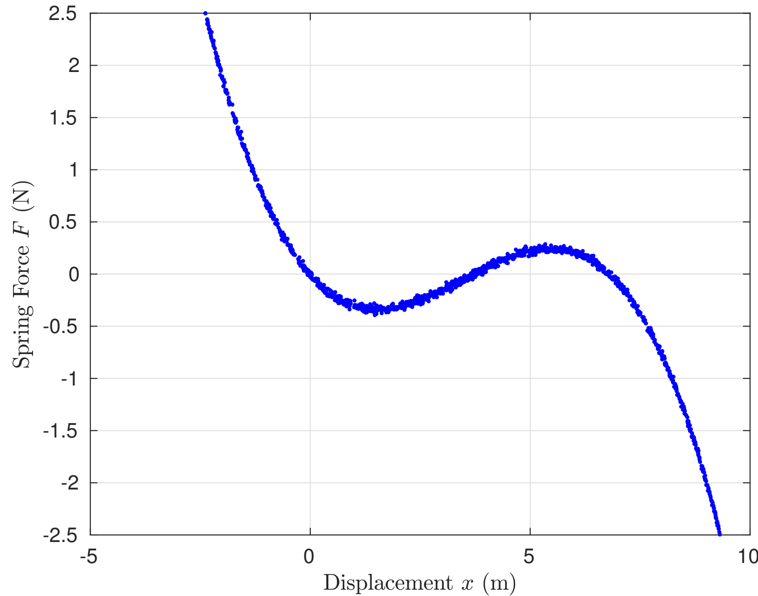



**Figure 5.11:** A dataset describing the force characteristic of the imagined nonlinear spring. There are three equilibrium points where $F = 0$, $\big[x = 0$ (stable), 3.75 (unstable), 6.80 (stable)$\big]$.

The purpose of this final case study is to investigate how neural network models and controllers may be used and analysed together in a closed loop. Using the insights gained from Case III, a hybrid model with a 1-dimensional data-driven component was considered.

Consider a frictionless nonlinear spring, with a measured force characteristic as shown in figure 5.11. The spring has 3 equilibrium points where $F = 0$, two of which are stable. It is assumed that a force can be applied to the spring, i.e through a linear actuator. An interesting control objective is to force the system to stay at its unstable equilibrium point. The equation of motion is:

$$\ddot{x} = k(x) + u(x) \tag{5.9}$$

Where $x$ is the displacement of the spring. Defining $x_1 = x$ and $x_2 = \dot{x}$, the following set of ODEs is constructed:

$$\dot{\boldsymbol{x}} = \begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} x_2 \\ k(x_1) + u(x_1) \end{bmatrix} \tag{5.10}$$

The previous case studies showed that simply training a neural network on such a system can lead to unnecessary complexity, as the network is forced to learn the simple relation $\dot{x_1} = x_2$. Instead, a neural network was trained on the spring force data alone, and then embedded in equation (5.10), replacing $k(x_1)$. This neural network model is written as $\hat{k}(x_1)$.

The neural network was given 2 hidden layers (with 50 and 25 nodes respectively) and was trained on the force data for 50 epochs, using the ADAM optimiser and a learning rate of 0.001.

Inserting the neural network into equation (5.10) yields a **hybrid model**, where $\dot{x_1}$ is defined explicitly as an equation and $\dot{x_2}$ is given by the network. The network was then converted to its PWA representation for analysis, shown in figure 5.12a. Using the formulation from the background (section 2.3.5), the function computed by the network can be written as:

$$\hat{k}(\boldsymbol{x}) = a_i x_1 + f_i = \begin{bmatrix} a_i & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}, \quad x_1 \in \Omega_i \tag{5.11}$$

Where $a_i$ is the slope in region $\Omega_i$ and $f_i$ is the offset. The regions $\Omega_i$ are 1-dimensional intervals. The hybrid model (5.10) can now be fully expressed as a PWA function:

$$\dot{\boldsymbol{x}} = \begin{bmatrix} 0 & 1 \\ a_i & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 0 \\ f_i \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} u(\boldsymbol{x}), \quad \boldsymbol{x} \in \Omega_i \tag{5.12}$$

The dimension of the regions $\Omega_i$ had to be lifted to include $x_2$ and $u$ in their domain. This was done by adding $x_2$ and $u$ as free variables to the H-representation of each region, by adding zero columns to the H-matrix in the $x_2$ and $u$ positions. The resulting linear regions in $\mathbb{R}^3$ are shown in figure 5.14a. Note that although the regions have been extended to $\mathbb{R}^3$, the total number of regions has not changed. Due to their dependence on $x_1$ only, the partition is far simpler than the previous models that were considered in the previous case studies.

The simplest approach to creating a controller for this system is to perform a plant inversion (also known as **feedback linearisation**). This involves canceling out the nonlinear dynamics of the system and inserting the desired dynamics. For example, assume the desired dynamics are that of a linear spring with its equilibrium point at $x_1 = 3.75$. This is achieved by setting $u(x) = -K_p(x_1 - 3.75) - \hat{k}(x)$, where $K_p$ is a parameter. The controller can then be expressed as:

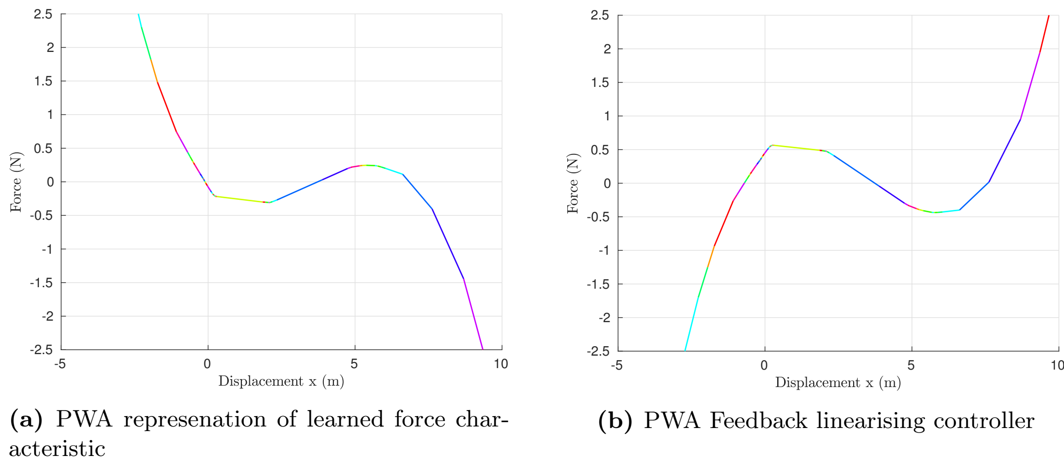
$$u(\boldsymbol{x}) = \begin{bmatrix} -(K_p + a_i) & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + 3.75K_p + f_i \tag{5.13}$$

The closed loop dynamics have then been set to that of a linear spring.

$$\ddot{x} = k(x) + u(x) = -k'(x_1 - 3.75) \tag{5.14}$$

Note that this controller will only cause stable oscillations, as there is no damping term. This is addressed at the end of the study.

The direct use of feedback linearisation on the PWA representation of $\hat{k}(x)$ will result in a controller that has the same linear regions (figure 5.12b). Therefore, when constructing the closed loop system using the controller and model, no additional linear regions will be created.

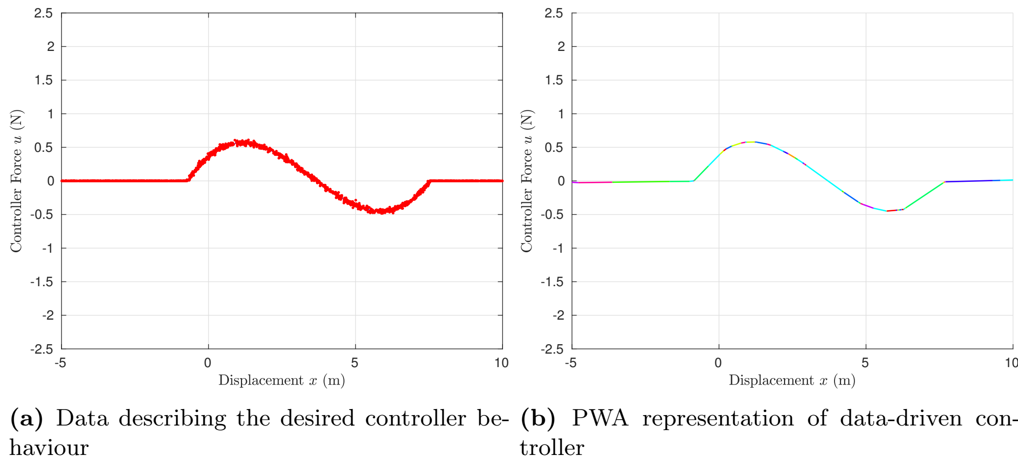



**(a)** PWA represenation of learned force characteristic

**(b)** PWA Feedback linearising controller

**Figure 5.12:** Feedback linearisation was used on the PWA model of the spring force characteristic seen in (a) to create a PWA controller with the same linear regions, shown in (b). This controller relies heavily on the accuracy of the model. If the model is inaccurate, then this controller introduces unwanted nonlinearities.

Feedback linearisation is simple and intuitive, but it is also relies heavily on the accuracy of the model. Robust control or data-driven methods such as reinforcement learning or genetic optimisation may be used to obtain controllers that are more resilient to model error. For example, running a reinforcement learning algorithm with randomly perturbed models will "teach" agent to stabilise a larger range of possible models, making it more robust. However, this will make the closed loop system more complex, as the linear regions of the controller will no longer match that of the model. Therefore, when computing the closed-loop system, many additional regions will be generated by the overlaps between the model and controller regions. To demonstrate this, a data-driven controller was created.




**(a)** Data describing the desired controller behaviour

**(b)** PWA representation of data-driven controller

**Figure 5.13:** The data in (a) was created by subtracting the data seen in figure 5.11 from the function $-K_p x_1$, where $K_p$ was chosen to be 0.2, creating a dataset that describes the behaviour of a linearising feedback controller. The two ends of the dataset were also clipped to zero. A neural network was then trained on this data, resulting in a data-driven controller. The PWA form of this network can be seen in (b), where the output is colour-coded according to the linear regions of the network.

While the use of more advanced techniques such as genetic optimisation and reinforcement learning was

determined to be out of the scope of this work, the situation was emulated by creating a dataset that describes the desired controller behaviour, as shown in figure 5.13a. This data was generated by performing feedback linearisation on the force characteristic data itself, and clipping the ends to zero. A neural network with the same architecture as the $\hat{k}(x)$ network was trained on the controller dataset for 100 epochs using the ADAM optimiser with a learning rate of 0.001. The PWA representation can be seen in figure 5.13b.

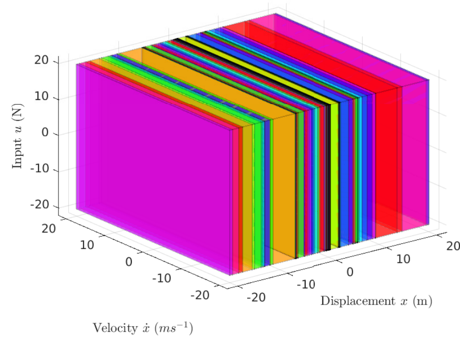
The data-driven controller accomplishes the same task as the controller obtained from feedback linearisation, but its linear regions are completely different. The additional complexity that this creates is shown in figures 5.14a, 5.14b, and 5.14c.

This effect is exacerbated when the regions vary in more dimensions. Additional linear regions were added to the data-driven controller introducing 5 additional switching boundaries wrt $x_2$. This increased the number of regions of the controller to 348, and the regions of the closed loop system soared to 1152 (see figure 5.15a). This highlights the importance of minimising the size of any data-driven components of a model, as well as selecting components that are defined on the same linear regions for maximum compatibility.

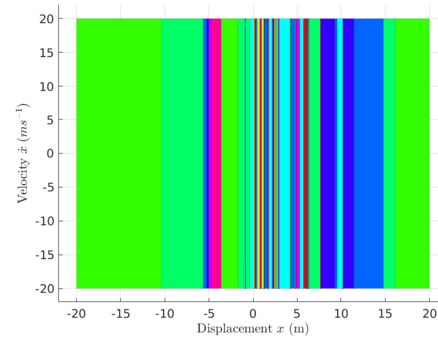
As a last note, the controllers discussed above do not control the system successfully, as the closed loop dynamics lack a damping effect. A damping term was added to the linearising feedback controller, so that $u(x) = -K_p(x_1 - 3.75) - \hat{k}(x) - dx_2$, where d is the damping coefficient. This was done by modifying the PWA representation of the data-driven controller (see (5.13)), so that:

$$u(\boldsymbol{x}) = \begin{bmatrix} k_i & -d \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + 3.75K_p - g_i, \quad \boldsymbol{x} \in \Omega_i \tag{5.15}$$

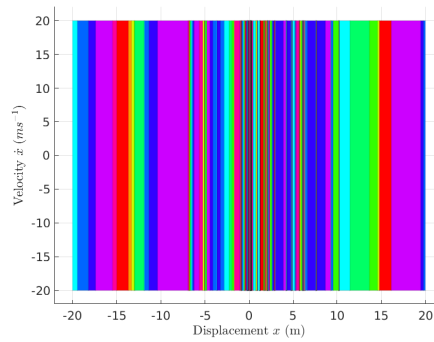
The resulting PWA controller and a simulation of the closed loop dynamics can be seen in figure 5.16a.




**(a)** Linear regions (135 total) of the hybrid model (5.10)

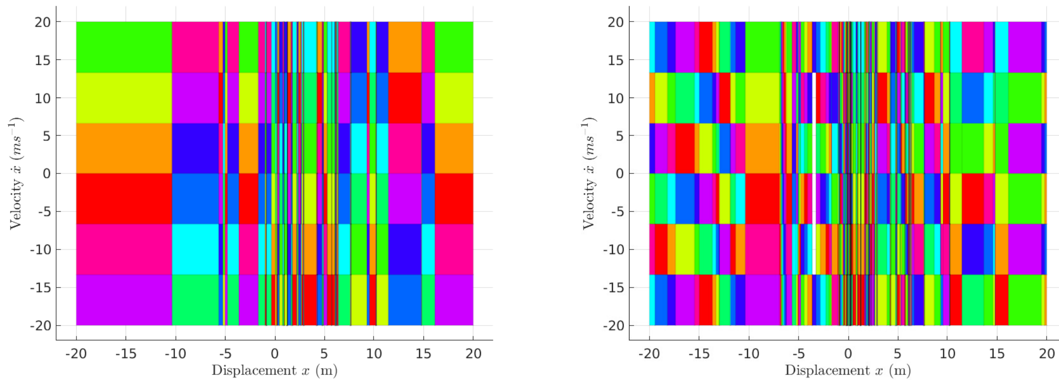



**(b)** Linear regions (58 total) of the data-driven controller




**(c)** Linear regions (192 total) of the closed loop system

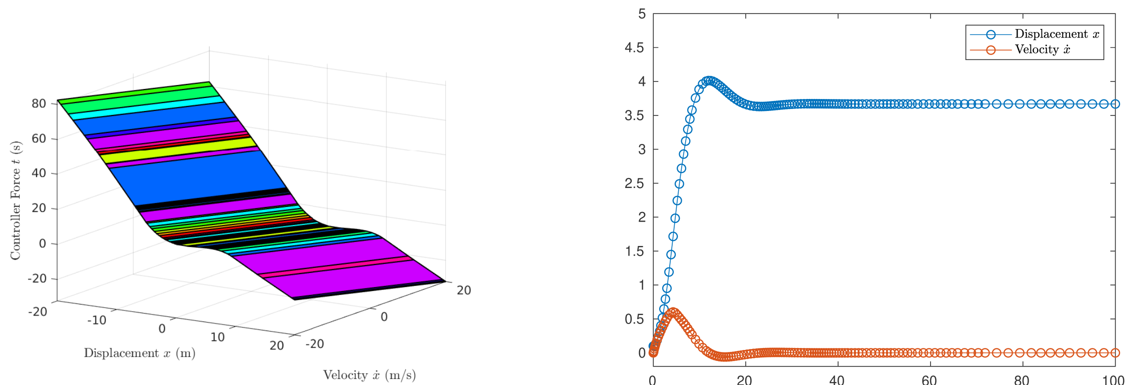
**Figure 5.14:** The dimension of the linear regions of the model (a) and controller (b) have been lifted without changing the number of regions. To see this, notice that only movements along the displacement axis will result in region transitions. The linear regions of the model and controller are combined to obtain the regions of the closed loop system. This process can be visualised as placing (b) at the bottom of (a) so that the axes line up, and then looking down from the top, superimposing the boundaries.




**(a)** Linear regions of data-driven controller with additional dependencies on $x_2$

**(b)** Linear regions of closed loop system with additional dependencies on $x_2$

**Figure 5.15:** Here the data-driven controller has been allowed to vary with the velocity $\dot{x}$ as well. 5 additional boundaries were artificially added to the controller (see left), resulting in a significantly more complex, augmented controller with 348 linear regions. Note that different dynamics were not actually added, so the behaviour will remain completely unchanged. The resulting closed loop system is now far more complex, with 1152 regions in total.




**(a)** Linearising feedback controller with damping effect

**(b)** Closed loop simulation of the spring with the damped controller

**Figure 5.16:** The linearising feedback controller was augmented with a damping effect, which can be seen as a linear slope along the $x_2$ axis. The closed loop system was simulated using the Dormand-Prince method (**ode45()** in MATLAB). The displacement converges to $x = 3.7$, as desired, and the velocity converges to 0.

## Comments

The main insights gleaned from this study are:

- The complexity of closed loop systems is vastly increased when both models and controllers are PWA systems defined on different linear regions. The added complexity increases with the dimension of the regions.
- More generally, a hybrid model may contain several data-driven components. If these components have a PWA representation, but are not defined on the same regions, then the total PWA form will be far more complex.

- It is relatively simple to modify PWA systems to include additional dynamics, even though libraries such as the MPT toolbox do not provide builtin functionality for this.

# 6 | Discussion

## 6.1 | Algorithm

The developed algorithm is novel in that it yields an exact, piecewise affine representation of neural networks that can open up whole new perspectives on the analysis of these popular, yet poorly understood models. The conversion can be performed on any network using piecewise affine activation functions such as ReLU, which is a popular choice due to its simplicity and its ability to mitigate the vanishing gradient problem (see section 2.3.6). Furthermore, the approach can be extended to handle networks with arbitrary branching structures. However, there are some clear limitations, to which the rest of this section is devoted.

Significant challenges include:

(a) The runtime of the algorithm scales linearly with the number of linear regions that the network has. Unfortunately, the number of regions also grows exponentially with respect to both the input dimension *and* the depth of the network. In practice, it was intractable to convert neural networks of significant size with input dimensions over four.

(b) One of the key subroutines in the algorithm, **repartition()**, involves finding the cells of a hyperplane arrangement, which is a challenging problem in its own right. This is made much worse by the fact that **repartition()** is called once per region found in all preceding layers.

(c) The set operations that the algorithm is based on involve a large number of (relatively small) LPs.

(d) The algorithm can be applied to a large variety of neural networks. However, the implementation only works with the Deep Learning Toolbox for MATLAB, while most research in deep learning is done with Python libraries such as Tensorflow.

**Item (a)** mentions the high runtime complexity of the algorithm, especially for networks with many inputs. The runtime aspect is very related to **item (b)**, as **repartition()** is a clear bottleneck in the algorithm. However, it should be noted that this does not mean that the algorithm is not practical! In many settings dimensionality reduction techniques such as **principal components analysis** (PCA) are used to compress the feature space containing the inputs for neural network models, retaining only a few principle variables / modes. The PWA conversion algorithm could then be applied to these reduced models.

Efforts were made to address **item (b)** by optimising the algorithm. The changes made to **repartition()** resulted in moderate speedups, especially for networks with wider layers. As mentioned, **repartition()** is called frequently; once per previously found region, for each layer in the network. Intuitively, this is wasteful because the hyperplane arrangement problem is solved completely independently in each case. However, the node boundaries of the next layer will be continuous, which is not taken advantage of by the current implementation.

An alternative approach would be to follow the node boundaries between regions, such that the algorithm only bisects the regions that it passes through. This could reduce the number of intersection checks performed, which is the main cost associated with **repartition()**. The procedure would be performed once per node in each layer, and could possibly be done in parallel.

There are some unanswered questions regarding this approach. Firstly, how should the process be initialised? Before the boundary can be followed, a starting region must be found. Any reduction in the number of intersection checks could be lost in the search for this initial region.

Secondly, once in a region, how can the relevant neighbouring regions be identified? This would involve finding the shared faces, which requires intersection checks with each constraint in the H-representation. The H-representation will only grow in size as the dimension of the problem increases. It is therefore unclear if this approach would reduce the total number of intersection checks.

Thirdly, how can the algorithm efficiently "move" from one region to the another once the relevant neighbours have been identified? This requires some careful thought about how the regions are represented/stored, so that it is simple to figure out which regions are adjacent.

Finally, how could such a method be run in parallel? Each node in a layer could be considered separately, but combining the results could be a challenge because the node boundaries themselves can intersect. In contrast, the developed algorithm runs **repartition()** once per previously found region, of which there could be 1000s or even 10000s. Therefore, the work can easily be divided up among i.e. 1000 cores, which could (theorectically) improve the runtime by a factor of 1000! This easy speedup was demonstrated in section 4.4 by running the algorithm on a 6 core machine, which improved the runtimes by a factor slightly lower than 6 (the loss is likely due to thermal regulators in the CPU). In contrast, the proposed method would be limited to the number of nodes in a neural network layer.

It is likely that the work could benefit from a hybrid approach, by selecting the most appropriate algorithm using some heuristic based on observed runtime data.

Alternative approaches to solving hyperplane arrangements might include:

- **Treating it as a search problem**: Every cell/region in the hyperplane arrangement can be defined by a unique "pattern" that represents which side of each hyperplane it is located. For example, in section 4.1.1 each region was assigned a color code corresponding to which nodes were active or not. This could also be expressed as a label such as "++-" or "110". Importantly, a move between adjacent regions can be made by moving across the shared boundary, which corresponds to flipping one of the bits in the pattern. Starting in some initial region, the algorithm could then perform a breadth first search by toggling each bit. Not every pattern will be valid however, so each combination would need to be verified by checking that the resulting set is not empty. Duplicates would also need to be avoided by removing redundant hyperplanes, which is an expensive operation. The reason this might be effective is that the adjacency of regions suggests that all of the valid patterns are in a sense "close", huddled together in a subset of all of $2^n$ combinations. it remains to be seen whether this intuition holds in higher dimensions.
- **Find hyperplane intersections instead**: The hyperplanes of the arrangement will intersect with other at vertices. There are far fewer of these vertices than there are regions. Yet, all regions must be adjacent to at least one vertex. Additionally, all regions that share a vertex are separated only by the constraints that intersect at that vertex, allowing the patterns for the regions to be easily found. Looking at figure 4.3, this could be visualised as finding all of the points where the lines intersect, and then rotating around each point to find the regions. Vertices share neighbouring regions, so care would have to be taken to avoid duplicates. An approach like this is discussed in Gerstner and Holtz (2006).

**Item (c)** points out that the algorithm makes a very high number of calls to a single-threaded LP solver. Additionally, the LPs are quite small, which suggests that simply initialising and returning from the solver incurs a large overhead relative to the cost of actually solving the LPs, though this has not been measured. The algorithm could therefore gain significantly from the use of a parallel LP solver that runs on the GPU (which typically have 100s of low power cores).

Such solvers exist, although they are not yet commonplace due to the difficulties involved in programming for GPU targets. There is also significant overhead involved in transferring data between working CPU and GPU memory, which would need to be taken into consideration. Recently however, there has been increased interest in the area. Examples of GPU implementations of the simplex method and the revised simplex method can be found in Spampinato and Elstery (2009), Lalami et al. (2011), and Bieling et al. (2010).

GPU solvers aside, the choice of single-threaded solvers is also important (a user may not have access to a

GPU). By default, the MPT toolbox is configured to use the MATLAB implementation of the dual-simplex method (part of the **linprog()** function). Despite the small size of the LPs, switching to another solver such as CPLEX could yield moderate speedups.

Finally, **item (d)** criticises the usability of the work due to its implementation in MATLAB. As mentioned in section 3, the decision to implement in MATLAB was motivated by the existence of the MPT toolbox, which greatly facilitated experimentation in control applications (as seen in the case studies). However, the vast majority of deep learning research is done in Python. There are two ways forward, first of which is to reimplement the algorithm in Python. This comes with its own challenges, as there is no well developed alternative to the MPT toolbox for Python, and efficient, scientific computing in Python can be challenging unless there is a dedicated, optimised library available.

The second approach is to utilise the ONNX standard for deep learning models, which allows import/export of models between different platforms. In the authors experience though, the MATLAB routines for importing ONNX models are unreliable and have certain limitations. The better choice would be to parse the ONNX files themselves, as they neatly describe the networks as a directed acyclic graph of well defined matrix operations[1]. This would allow the models to be designed and trained using arbitrary software, and parsed directly in MATLAB. This would also involve extending the algorithm to handle more complicated operations and network architectures; i.e. convolutional layers and branching networks.

## 6.2 | Case studies and future applicability of the work

Current methods for the analysis of PWA systems struggle with systems that have a high number of modes. The complexity only increases with the dimension of the system, limiting the application of the PWA conversion algorithm to larger problems. This is in direct conflict with the key perceived strength of neural networks; the ability to handle extremely high-dimensional and non-linear data.

In this sense, the case studies are limited in scope, as they only considered relatively small neural networks. However, the case studies show that neural networks often complicate the problem by overpartitioning the state space in an unstructured way. More interpretable hybrid models can then be obtained by reducing the size of the data driven components as much as possible without sacrificing performance.

They highlight where future work might be directed. The key insights gained are:

(a) While neural networks may be used in modeling contexts, they do not follow conservation laws or symmetries.

(b) As seen in Case II, applying standard stability verification strategies such as the Lyapunov method to non-trivial neural network models is difficult due to their small-scale irregularity.

(c) While neural networks display some ability to partition the state space in a useful way (case II), this does not seem to be the case for higher dimensional problems (case III).

(d) The PWA form of neural network models is not directly useful for online MPC purposes.

(e) Care must be taken when using multiple neural network models/controllers together, to avoid an explosion in the linear region complexity.

**Point (a)** has consequences for the validity and usefulness of neural network models. For example, when simulating the dynamics learned by the network in Case II it was observed that the trajectory was not symmetric. This is in spite of the fact that the data the network was trained on had no noise, and thus represented the exact mapping between $\boldsymbol{x}$ and $\dot{\boldsymbol{x}}$. This presents an interesting question: Is it possible to encode conservation laws or symmetries in the architecture of the network itself? If yes, then this would greatly increase the applicability of neural networks in modeling contexts. It is also likely that this would help

[1] Read more at github.com/onnx/onnx

speed up the training of these neural network models, as the enforcement of conservation laws or symmetries would act as a constraint on the parameters of the network. In principle, this should help guide the training process towards realistic solutions.

There have been recent efforts to make neural networks more suitable for modeling real systems Grzeszczuk et al. (2000), scenes of multiple objects Battaglia et al. (2016); Chang et al. (2016), and even the potential fields of proteins Behler (2014).

**Point (b)** was observed in Case II, after an attempt at applying the Lyapunov method to a neural network model. With standard training methods, it is almost impossible to guarantee that the function computed by the network will be regular at the small scale. This is because the networks will pick up on the noise inherent to datasets. Also, the network will generally have many small linear regions that have little to no effect on the overall shape of the output. This means this regions will not contribute much to the error of the network during training, such that the backpropagation process will be unable to smooth them out.

Avoiding small regions in the network is one way to mitigate the problem, but it is not at all obvious how to do this well. For example, checking the size of regions during training and performing some kind of regularisation would make the backpropagation process prohibitively expensive. Smoothing the PWA representation by merging small regions with their neighbours could achieve this while keeping the output roughly the same, but then the exactness of the representation would be lost. This might be acceptable, assuming that the output is not changed significantly.

The solutions discussed for point (a) might also be relevant here, as imposing conservation laws / symmetries on the network could help regularise its output.

If the representation or the network cannot be changed, perhaps the methods can. The Lyapunov method works by showing that the energy associated with the state is always being dissipated in the absence of input. One possibility then is to relax the method for stability verification, permitting energy increases at the small scale, perhaps when the state energy is below a certain threshold. This could allow the model function to be slightly irregular. It might also be possible to include the threshold as a decision variable in the optimisation problem, which would provide a measure of how "well behaved" the system is.

**Point (c)** refers to the contrast seen between figures 5.8 and 5.9 from Case III. This suggests neural networks tend to create unnecessarily complex partitions of the state space. Recall that an attempt was made to minimize the size of the larger network. Yet, the simulation results seem to show almost no difference between the *output quality* of the two networks.

In contrast, the neural networks with lower input dimension (Cases I and II) seem to align their linear regions in a more sensible fashion, forming shapes that can easily be adjusted to the required shape[2]. Why is this?

The linear regions of the simple network from Case III (figure 5.8) may help answer this question. It was previously noted that the boundaries formed an 'X' shape. Compare this to the Case I network, where all of the boundaries have congregated to the middle in order to produce a flat output (to express $\theta = \dot{\theta}$. They are not superimposed however, as they form a collection of very thin 'X's.

It is reasonable to conclude that this arrangement serves to cancel out the contributions of the boundaries on both arms of the 'X'. In this sense, the boundaries are locked in a balancing act; moving only one arm of the 'X' will break the balance and result in an output that doesn't match the data. Yet, it is intuitive in both Case II and III that the arrangement is redundant, and the network would achieve a perfect output if it just superimposed the boundaries to form a single crease.

This hypothesis could help explain the remarkably complex structure seen in Case III, figure 5.9. If many of the node boundaries are canceling each other out in order to produce a flat output, then breaking the balance will only yield worse performance. Such arrangements are thus in some sense stable, and can be related to the idea of a local minimum in optimisation.

[2]The boundaries can be understood as *creases* in the output, where it is allowed to bend

It may be enlightening to reframe the training process of a neural network with piecewise activations as the movement of these node boundaries through the state space. For example, to mitigate the redundant 'X' arrangements described above, an attractive force between the orientation of boundaries could be imposed. This could help address points (a) and (b) by removing small regions that exist between node boundaries that are close, but not superimposed. It is also conceivable that this could give rise to an adaptive type of neural network, where nodes can be merged when they get too close. Such a "self-simplifying" model could be extremely useful.

**Point (d)** concerns the observations made in Case III. It was demonstrated that creating or running optimal controllers on PWA systems is expensive, especially when the number of regions is large. This is to be expected, as the piecewise dynamics become quite complex and expensive to evaluate when solving the dynamic optimisation problem. When online MPC is too expensive, it is common to compute the explicit MPC solution (see 2) by solving a multiparametric optimisation problem. This was not considered in the case study, as this is a more expensive computation than online MPC, which was already beyond the available computational resources.

However, this may still be tractable. In Chapter 3 of Grancharova and Johansen (2012), an algorithm for computing approximate explicit MPC solutions for nonlinear systems is presented. The optimisation problem is mp-NLP, but is approximated by solving mp-QPs in different regions of the state space. This is done by recursively partitioning the state space into regions (thus controlling the accuracy of the local approximation), and solving the mp-QP for each sub region. The solution is stored in this tree structure, and then later accessed by searching through the tree. The PWA conversion algorithm also makes use of recursive partitioning, such that the resulting PWA function could be stored as a tree-like structure of regions. The latter part of the NMPC algorithm could then be applied to this partition, by solving locally approximating mp-QPs yielding an explicit MPC controller for the system.

Regarding **Point (e)**, Case IV showed that utilising both data driven models and controllers together can introduce additional complexity to the resulting closed loop system, as all intersections between the 2 different PWA functions have to be taken into account. This effect scales with the dimension, increasing the number of ways that the regions can intersect with each other.

The most obvious solution to avoiding this complexity increase is to ensure that the PWA model and the PWA controller have exactly the same linear regions. As shown in section 4.1.2, adding layers with no activation to a neural network will not modify its linear regions. Therefore, the linear regions of a neural network can easily be duplicated by copying and fixing the weights of all the layers with activation. An additional layer with no activation can then be added and trained to match the desired data.

This has the advantage of having to train fewer parameters, and being less sensitive to parameter changes (allowing for a higher learning rate). However, it remains unclear how much control the last layer has over the output of the network, as the last layer only performs an affine transformation on the function computed by the previous layers.

One last comment is that the case studies did not cover the robustness of the data-driven models. In Chapter 4.2 of Lazar (2006), it is shown that a seemingly stable system with a valid Lyapunov candidate has zero robustness, implying that even the smallest disturbance will cause it to become unstable. This is a consequence of allowing the Lyapunov function to be discontinuous. Robustness and sensitivity analysis for neural networks is also an area of significant research interest. Combining existing methods for robust PWA systems and neural networks could be a fruitful line of research.

# 7 | Conclusion

To reiterate, the guiding questions given at the start of section 2 were:

- *How can a neural network be converted to its piecewise affine form?*
- *How can the stability of a dynamical model based on a neural network be verified?*
- *Is it practical to design controllers for dynamical neural network models?*

The first guiding question was successfully answered with the implementation of a reasonably efficient algorithm that can perform the conversion. Furthermore, multiple improvements in terms of efficiency and extensions have been outlined in the discussion. Results demonstrating conversions of neural networks with up to four dimensions were reported, the largest of which had around 10000 linear regions. A parallelised version of the algorithm was able to perform this conversion in around a minute on a standard desktop computer. With more computational resources and time, it is clear that much larger networks will be able to to be analysed. Using the PWA conversion algorithm together with dimensionality reduction techniques is an approach that shows great promise for the study of complex systems that resist analysis.

The stability of a simple LTI system using a neural network controller was verified using the Lyapunov method. However, the same approach did not generalise to more complex systems, due to the typically irregular output of neural networks. The answer to the second guiding question is therefore tentative, and two directions for future work in this area were outlined. The first solution is to impose some form of regularisation on the neural network, either during training or by modifying its architecture. The second solution proposed a relaxed Lyapunov method.

The third guiding question was approached in two different ways. First, an attempt was made to control the PWA form of neural network models using MPC. The computations involved with this turned out to be very expensive and impractical on a desktop computer. Despite this, links were drawn between existing explicit nonlinear MPC methods and the recursive nature of the PWA form of neural network models. Combining the two methods could yield a cheap, near-optimal controller for neural network models. The second approach considered the possibility of using data-driven controllers based on neural networks as well. This highlighted the importance of choosing controllers that have linear regions that are *compatible* with those of the neural network, in order to avoid a significant increase in complexity when obtaining the closed loop system. A simple feedback linearisation approach was also undertaken, although concerns about the robustness of the resulting closed loop system were raised. Robustness in the context of neural network models is an important topic that must be investigated in the future.

This work has been a novel approach to the study of neural networks. Although more questions were found than answered, many insights into the internal complexity of these opaque models have been gleaned in the process. It is the hope of the author this will provide inspiration for new avenues of research into the safe and responsible use of neural networks for modeling and control.

# Bibliography


M. Baotic. Polytopic computations in constrained optimal control. *Automatika*, 50(3-4):119–134, 2009.

C. B. Barber, D. P. Dobkin, D. P. Dobkin, and H. Huhdanpaa. The quickhull algorithm for convex hulls. *ACM Transactions on Mathematical Software (TOMS)*, 22(4):469–483, 1996.

P. Battaglia, R. Pascanu, M. Lai, D. J. Rezende, et al. Interaction networks for learning about objects, relations and physics. In *Advances in neural information processing systems*, pages 4502–4510, 2016.

J. Behler. Representing potential energy surfaces by high-dimensional neural network potentials. *Journal of Physics: Condensed Matter*, 26(18):183001, apr 2014. doi: 10.1088/0953-8984/26/18/183001. URL `https://doi.org/10.1088%2F0953-8984%2F26%2F18%2F183001`.

A. Bemporad, F. Borrelli, M. Morari, et al. Model predictive control based on linear programming˜ the explicit solution. *IEEE transactions on automatic control*, 47(12):1974–1985, 2002.

J. Bieling, P. Peschlow, and P. Martini. An efficient GPU implementation of the revised simplex method. In *2010 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW)*, pages 1–8, April 2010. doi: 10.1109/IPDPSW.2010.5470831.

M. B. Chang, T. Ullman, A. Torralba, and J. B. Tenenbaum. A compositional object-based approach to learning physical dynamics. *CoRR*, abs/1612.00341, 2016. URL `http://arxiv.org/abs/1612.00341`.

S. Chen, K. Saulnier, N. Atanasov, D. D. Lee, V. Kumar, G. J. Pappas, and M. Morari. Approximating explicit model predictive control using constrained neural networks. In *2018 Annual American Control Conference (ACC)*, pages 1520–1527, June 2018. doi: 10.23919/ACC.2018.8431275.

T. Q. Chen, Y. Rubanova, J. Bettencourt, and D. K. Duvenaud. Neural ordinary differential equations. In *Advances in Neural Information Processing Systems*, pages 6571–6583, 2018.

B. C. Csáji. Approximation with artificial neural networks. *Faculty of Sciences, Etvs Lornd University, Hungary*, 24:48, 2001.

R. Eldan and O. Shamir. The power of depth for feedforward neural networks. *CoRR*, abs/1512.03965, 2015. URL `http://arxiv.org/abs/1512.03965`.

K. Fukuda et al. Frequently asked questions in polyhedral computation. *Swiss Fedral Institute of Technology*, 2004.

T. Gerstner and M. Holtz. Algorithms for the cell enumeration and orthant decomposition of hyperplane arrangements. *University of Bonn*, 2006.

A. Grancharova and T. A. Johansen. *Explicit nonlinear model predictive control: Theory and applications*, volume 429. Springer Science & Business Media, 2012.

R. Grzeszczuk, D. Terzopoulos, and G. Hinton. *NeuroAnimator: fast neural network emulation and control of physics-based models.* University of Toronto, 2000.

M. Herceg, M. Kvasnica, C. Jones, and M. Morari. Multi-Parametric Toolbox 3.0. In *Proc. of the European Control Conference*, pages 502–510, Zürich, Switzerland, July 17–19 2013. `http://control.ee.ethz.ch/~mpt`.

H. K. Khalil. *Nonlinear systems*, volume 3. Pearson, 2002.

M. E. Lalami, D. El-Baz, and V. Boyer. Multi GPU implementation of the simplex algorithm. In *2011 IEEE International Conference on High Performance Computing and Communications*, pages 179–186, Sep. 2011. doi: 10.1109/HPCC.2011.32.

M. Lazar. *Model predictive control of hybrid systems: Stability and robustness.* PhD thesis, Eindhoven University of Technology, 2006.

I. J. Leontaritis and S. A. Billings. Input-output parametric models for non-linear systems part i: deterministic non-linear systems. *International Journal of Control*, 41(2):303–328, 1985. doi: 10.1080/0020718508961129. URL `https://doi.org/10.1080/0020718508961129`.

J. Löfberg. Yalmip : A toolbox for modeling and optimization in matlab. In *In Proceedings of the CACSD Conference*, Taipei, Taiwan, 2004.

G. F. Montufar, R. Pascanu, K. Cho, and Y. Bengio. On the number of linear regions of deep neural networks. In *Advances in neural information processing systems*, pages 2924–2932, 2014.

J. Nocedal and S. Wright. *Numerical optimization.* Springer Science & Business Media, 2006.

R. Pascanu, G. Montufar, and Y. Bengio. On the number of response regions of deep feed forward networks with piece-wise linear activations. *arXiv preprint arXiv:1312.6098*, 2013.

M. Raghu, B. Poole, J. Kleinberg, S. Ganguli, and J. S. Dickstein. On the expressive power of deep neural networks. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ICML'17, pages 2847–2854. JMLR.org, 2017. URL `http://dl.acm.org/citation.cfm?id=3305890.3305975`.

J. B. Rawlings and D. Q. Mayne. *Model predictive control: Theory and design.* Nob Hill Pub. Madison, Wisconsin, 2009.

L. Ruthotto and E. Haber. Deep neural networks motivated by partial differential equations. *arXiv preprint arXiv:1804.04272*, 2018.

C. Said. Video shows uber robot car in fatal accident did not try to avoid woman. `sfgate.com/business/article/Uber-video-shows-robot-car-in-fatal-accident-did-12771938.php`, March 2018.

T. Serra, C. Tjandraatmadja, and S. Ramalingam. Bounding and counting linear regions of deep neural networks. In *International Conference on Machine Learning*, pages 4565–4573, 2018.

H. T. Siegelmann, B. G. Horne, and C. L. Giles. Computational capabilities of recurrent narx neural networks. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 27(2):208–215, April 1997. ISSN 1083-4419. doi: 10.1109/3477.558801.

D. G. Spampinato and A. C. Elstery. Linear optimization on modern gpus. In *2009 IEEE International Symposium on Parallel Distributed Processing*, pages 1–8, May 2009. doi: 10.1109/IPDPS.2009.5161106.

R. P. Stanley. *An Introduction to Hyperplane Arrangments.* University of Pennsylvania, 2006.

M. Telgarsky. Representation benefits of deep feedforward networks. *CoRR*, abs/1509.08101, 2015. URL `http://arxiv.org/abs/1509.08101`.

T. Zaslavsky. Facing up to arrangements: Face-count formulas for partitions of space by hyperplanes. *Memoirs of the American Mathematical Society*, 154, 01 1975. doi: 10.1090/memo/0154.

# Appendices

# A | Methods in Computational Geometry

Here the operations described in section 2.3.4 are described in more detail.

- **Checking if a polyhedron is empty**: It may be that the H-representation describes an empty set. This may be checked by performing a *feasibility LP* (LP with no objective). If no feasible point is found, the set is empty.
- **Intersection of 2 convex polyhedra**: This can be done simply by vertically concatentating the H-representations of the two polyhedra. The result will always be convex. A polyhedron can also be intersected with a collection of polyhedra by considering each intersection individually, and then taking the union.
- **Union of 2 convex polyhedra**: This may not result in a convex polyhedron. If not, then the region must be described by using a collection of polyhedra. If the result *is* convex however, then the union can be found as a single convex polyhedron by computing the convex hull Barber et al. (1996).
- **Set difference of 2 polyhedra** ($P/Q$: This can be calculated as the intersection between $P$ and the complement of $Q$. However, the complement is calculated using the set difference. Instead, a new collection of polyhedra can be made by taking each half-space in the H-representation of $Q$ and flipping it. The union of the polyhedra in this new collection will be equal to the complement of $Q$, but they will overlap. This does not matter though, as the collection can now be intersected with $P$.
- **Complement of a polyhedron**: This can be computed as the set difference between $\mathbb{R}^d$ and the polyhedron. This cannot be easily done within the H-representation, as the result will not be a convex polyhedron and therefore not H-representible. However, the result can be expressed as a collection of polyhedra.
- **Chebyshev centre of a polyhedron**: The Chebyshev ball is the largest ball that may be fit inside the polyhedron. This may be found as the solution to a cheap LP (note that the centre may not be unique!).
- **Simplifying the H-representation**: Some of the half-spaces in the H-representation may be redundant. This may be done by solving an LP for every row in the H-representation, eliminating the rows that fail. To reduce the number of LPs required, heuristics such as bounding box comparisons and random raycasting may be used.

# B | Algorithms for PWA systems

## B.1 | Computing the transition map for a PWA system

The map consists of a square adjacency matrix $\boldsymbol{T}$ that contains all possible transitions between regions, and a collection of the subregions $\mathcal{R}$ where the transitions occur. The subregions are indexed in the same manner as the adjacency matrix, such that the subregion of $\Omega_i$ that enters $\Omega_j$ is $R_{ij}$. The transition map may be found with the following procedure:

1. Iterate through all pairs of regions $(\Omega_i, \Omega_j)$
2. Computing the inverse affine map $(\boldsymbol{A}_i \boldsymbol{x}_t + \boldsymbol{f}_i)$ for $\Omega_j$
3. Find the intersection of the result and $\Omega_i$. This is the subregion of $\Omega_i$ that will enter $\Omega_j$ on the next timestep.
4. If the intersection is empty, set $T_{ij} = 0$, and $R_{ij} = \emptyset$. If it is not empty, then set $T_{ij} = 1$, and $R_{ij} = (\boldsymbol{A}_i \boldsymbol{x}_t + \boldsymbol{f}_i) \cap \Omega_i$.

## B.2 | Verifying that a set of polyhedral regions is positive invariant

The positive invariantness of a PWA system can be verified by inspecting its transition map.

1. Start with $\Omega_i = \Omega_1$
2. Using the transition map, find the union of all transition regions within $\Omega_i$
3. If the transition regions do *not* cover $\Omega_i$, the set is **not positive invariant**
4. Otherwise, get the next region $\Omega_i + 1$ and go to point 2. If there are no more regions, then the set is **positive invariant**.

In other words, if there is a subregion of $\Omega_i$ that is not accounted for by the transition map, then that subregion will transition to a set that is *outside* of the known regions. The set can therefore not be positive invariant.

## B.3 | Finding the maximal positive invariant set

The maximal positive invariant set for some PWA system can be computed by repeatedly computing the backward reachable set, starting with some initial region. If the backwards reachable set converges, then the maximal positive invariant set has been found. Any parts of the backwards reachable set that leave the domain of the PWA function are removed. This procedure thus assumes that the domain of the PWA function is finite. State constraints may also be used when detecting whether part of the backwards reachable set has become invalid. The procedure is as follows:

1. Let $\boldsymbol{x}_{t_1} = f(\boldsymbol{x}_t)$ be an autonoumous PWA system with a domain composed of the regions $\Omega_i$.

2. Start with $\mathcal{R} = \{\Omega_i\}$. Alternatively, start with the state constraints $\Xi$.
3. Compute the backward reachable set $\mathcal{R}^+$ of $\mathcal{R}$ according to $f(\boldsymbol{x}_t)$.
4. Find the intersection $\mathcal{R}^+ \cap \Xi$, and compare it to $\mathcal{R}$. If they are the same, $\mathcal{R}$ is the **maximal positive invariant set**.
5. Otherwise, set $\mathcal{R} \leftarrow \mathcal{R}^+$ and go to step 3.

In summary, the procedure simulates the system backwards, clipping away the regions that leave the valid area. After some iterations, only regions that comprise a positive invariant set will remain. If the remaining regions stay unchanged upon further iterations, then the positive invariant set cannot grow any further, and the maximal positive invariant set has been found.