Max-Min-Plus-Scaling Neural Networks to Approximate Continuous Piecewise Affine Model Predictive Control Laws

Bouke Stoelinga

Master of Science Thesis

TU Delft Delft University of Technology

Delft Center for Systems and Control

Max-Min-Plus-Scaling Neural Networks to Approximate Continuous Piecewise Affine Model Predictive Control Laws

Master of Science Thesis

For the degree of Master of Science in Systems and Control at Delft University of Technology

Bouke Stoelinga

October 10, 2023

Faculty of Mechanical, Maritime and Materials Engineering (3mE) · Delft University of Technology

Delft University of Technology
Department of
Delft Center for Systems and Control (dcsc)

The undersigned hereby certify that they have read and recommend to the Faculty of Mechanical, Maritime and Materials Engineering (3mE) for acceptance a thesis entitled

Max-Min-Plus-Scaling Neural Networks to Approximate Continuous Piecewise Affine Model Predictive Control Laws

by

Bouke Stoelinga

in partial fulfillment of the requirements for the degree of

Master of Science Systems and Control

Dated: October 10, 2023

Supervisor(s):

Dr.ir. A.J.J. van den Boom

Ir. K. He

Reader(s):

Prof.dr.ir. B.H.K. De Schutter

Abstract

This thesis extensively examines the influential factors affecting the performance of approximations of Model Predictive Control (MPC) control laws using neural networks. MPC is a control strategy that solves an optimization problem at each timestep. This problem can be computationally complex and could be too slow to compute for online control. Sometimes an explicit solution for MPC exists, but this can become very large in memory and is not always available. That is why approximations with neural networks might offer a benefit. Under certain conditions, the explicit solution yields a piecewise affine (PWA) control law. A PWA model class is equivalent to the so-called Max-Min-Plus-Scaling (MMPS) model class, which is a generalization of max-plus and min-plus algebra. Neural networks are made up of neurons, which make use of activation functions. A feed-forward neural network with some specific activation function can yield an MMPS function. This inspires us to research the use of different activation functions in approximating MPC control laws. Additionally, we investigate different sampling strategies and the use of max-plus and min-plus layers in neural networks.

We do this by setting up different PWA and non-PWA control laws for two inverted pendulum systems and training several neural networks to approximate these control laws. We first observe a significantly better performance in approximating the PWA control laws compared to the non-PWA control laws. When varying the activation functions of the neural networks we find that for PWA control laws a MMPS activation function can offer a better performance, but it is not guaranteed for all MMPS functions. We also find that networks with custom max-plus layers can offer a similar performance on approximating control laws compared to networks with traditional layers. When investigating what sampling strategy is most beneficial we find comparable performance with a stratified sampling strategy and a uniform sampling strategy. Depending on what areas of the control law you want to capture with the most detail, you can choose the most viable sampling strategy. With this, we have researched various factors that influence the performance of approximations of MPC control laws. The thesis ends with a recommendation to research even more factors that might offer even better approximations.

Table of Contents

Preface and Acknowledgements

Dear reader,

After spending the last year focusing on this thesis and solving lots of challenges is finally time to hand in my final thesis and conclude my studies at Delft University of Technology. I am grateful for all the help I have received and I am proud of what I have achieved.

I would like to thank my supervisors Ton and Kanghui, who helped me with this topic that started as something I found curious, and they supported me in making a thesis out of it. Through many meetings and pointing out things that could be improved, they helped to get me to the best result.

I would also Like to thank Bart De Schutter for taking the time and effort to read and evaluate my thesis.

I would also like to thank my study friends from my association, with whom I spent numerous sessions working in Pulse and other places. The coffee and lunch breaks helped to make working on my thesis more fun and made writing this thesis a lot less lonely.

Finally, I would like to thank my family and my partner Casper, who supported me throughout the entire process. Their motivation and support helped me push through all the difficult challenges and made everything more enjoyable.

Delft, University of Technology
October 10, 2023

Bouke Stoelinga

Chapter 1

Introduction

1-1 Background

Model Predictive Control [1], [2] is a very useful control technique that uses a model of a system to compute an optimal solution. This optimal solution balances controller effort and state trajectories and can also handle constraints. For this, an optimization problem does need to be solved at each timestep. This optimization problem can in some cases be too complex to be solved in time for the next timestep. A solution that would not require an optimization problem at each step is explicit model predictive control [3]. This method uses multiparametric programming techniques to reduce the control law to a simple-to-evaluate function. These obtained control laws are sometimes Piecewise Affine (PWA), for example for a linear system with linear constraints. A disadvantage is that obtaining these explicit control laws can be nontrivial and they might have a large number of linear regions. However, if such an explicit control law is not available, approximations are an alternative.

The PWA form of some control laws is related to another area of study. In discrete event systems, PWA systems can be converted to a different class of functions, namely the max-min-plus-scaling (MMPS) systems [4]. The functions they use, PWA functions, and MMPS functions are also equivalent. These MMPS functions make only use of max, min, plus, and scaling operators. Related to them are the max-plus algebra and min-plus algebra, in which some nonlinear functions in conventional plus-times algebra are linear in their respective algebra.

Looking for approximations instead of exact explicit MPC control laws leads us to a technique for finding approximations of functions: artificial neural networks. We choose neural networks over other function approximation methods because their structure. With certain activation functions neural network can match the structure of a MMPS function. They are made up of neurons, and each neuron often has an activation function. What is remarkable here is that nowadays the most commonly used activation function Relu, and the standard feedforward structure of a neural network in combination with these activation functions, works as an MMPS function. MMPS functions also appear in a different type of neural network where max-plus and min-plus layers are implemented [5].

Neural networks have been used before to approximate control laws [6]–[8]. They use some sampling strategies: They simulate trajectories or take samples from an explicit control law. Several different neural networks structures have been used to approximate MPC control laws such as Long short-term memory networks [7], [9] or recurrent neural networks [10].

1-2 Problem description

We want to go further with estimating MPC control laws with neural networks. There are a few aspects that have yet to be thoroughly researched. These are namely:

- Differences in approximating PWA control laws and non-PWA control laws
- Difference in approximating PWA control laws with different MMPS and non-MMPS activation functions
- Difference in sampling strategies when approximating MPC control laws
- Application of max-plus and min-plus layers in approximating MPC control laws

To guide this research, some research questions are formulated.

1-2-1 Research Questions

The main question that arises is:

What are the various factors that influence the performance of neural network approximations of MPC control laws?

This main question will be answered with the help of the following subquestions:

- Is there a significant difference in approximating PWA MPC control laws compared to non-PWA MPC control laws?
- Does the max-min-plus-scaling structure of a neural network offer a benefit compared to other neural networks with other activation functions when approximating MPC control laws?
- Do max-plus and min-plus layers in a neural network offer better performance?
- What is an appropriate sampling strategy to obtain a satisfactory approximation of an MPC control law?

1-2-2 Approach

We will discover the various factors that influence the performance of neural network approximations of MPC control laws by surveying the literature and setting up experiments.

The first four chapters will cover the literature that helps us set up the experiments, we start in Chapter 2 by getting into different classes of functions and their equivalence. These classes

of functions will come back in Chapter 3 where we will find all the concepts we need to set up MPC controllers and obtain control laws. Chapter 4 will go into detail about neural networks that we need for approximating these control laws, what algorithms are used and what type of neural networks we want to test.

The rest of the chapters will contain our main contributions. Chapter 5 will describe the detailed approach of our experiments and how the research questions will be answered. We start by setting up MPC controllers for two different systems in Chapter 6 which will yield different control laws. Using these control laws we will train several different neural networks with different parameters. Afterwards, we will see how they perform by analyzing the validation loss and our own custom performance metric in Chapter 7.

Through the use of PWA and non-PWA control laws, we will find if there is a difference in performance for approximating these control laws. We will also test several different activation functions, including two MMPS activation functions to discover if they offer a benefit compared to other activation functions. Additionally, we also make use of max-plus and min-plus layers in some neural networks and analyze their performance. Finally, we will take subsets of our datasets to test different sampling strategies and find which find a satisfactory approximation.

Chapter 2

Max-Min-Plus-Scaling

This chapter will describe a specific group of functions called max-min-plus-scaling functions. They make use of max, min, plus, and scaling operations and are used in max-min-plus-scaling systems. These systems are equivalent to other forms and have canonical forms as well. The chapter ends by mentioning max-plus and min-plus algebra.

2-1 Max-Min-Plus-Scaling functions

Max-Min-Plus-Scaling (MMPS) functions are a group of functions that allow for non-smooth behavior while still being continuous. They are defined as:

Definition 2.1. *Max-min-plus-scaling functions. A max-min-plus-scaling (MMPS) function f mapping $\mathcal{R}^m \to \mathcal{R}$ for the variables $x_1, ..., x_m \in \mathcal{R}$ is defined by the grammar:*

$$f := x_i | \alpha | \min(f_k, f_l) | \max(f_k, f_l) | f_k + f_l | \beta f_k,$$

with $i \in \{1, 2, ...m\}$, $\alpha, \beta \in \mathbb{R}$ and f_k and f_l recursively defined MMPS functions. The symbol $|$ stands for or.

The set denoted by $\mathcal{R}$ is either $\mathbb{R}, \mathbb{R}_\varepsilon \stackrel{\text{def}}{=} \mathbb{R} \cup \{-\infty\}, \mathbb{R}_\top \stackrel{\text{def}}{=} \mathbb{R} \cup \{\infty\}$ or $\mathbb{R}_c \stackrel{\text{def}}{=} \mathbb{R} \cup \{-\infty, \infty\}$ depending on if only scaling operations, scaling and max operations, scaling and min operations or all scaling, min and max operations are used.

Example 2.2. *An example of an MMPS function in conventional notation:*

$$f(x) = \min\left(\max\left(x_1 + 3, x_2 - 4\right) + 5\max\left(x_3 + 6, 3\right), 4x_4\right).$$

These functions are often used to describe discrete event systems

2-1-1 Max-Min-Plus-Scaling systems

A discrete event system can be described with the state-space equations:

$$\begin{aligned} x(k) &= f_{\text{mmps}}\left(x(k-1), u(k)\right) \\ y(k) &= g_{\text{mmps}}\left(x(k), u(k)\right), \end{aligned} \tag{2-1}$$

where f_{mmps} and g_{mmps} are MMPS expressions, $x(k)$ is the state and $u(k)$ is the input. This is called an MMPS system [11] and is a modeling class of hybrid systems. These MMPS systems are equivalent to other classes of hybrid systems with certain conditions [4]. Such as Piecewise Affine (PWA) systems [12], Mixed Logical Dynamical (MLD) systems [13], Linear Complementary (LC) systems [14], [15] and Extended Linear Complementarity (ELC) systems [16].

2-1-2 Canonical forms of MMPS systems

There are 3 canonical forms of MMPS systems [11], [17]–[19].

Definition 2.3. *Conjunctive canonical form: Let $a_{i,j}$ be real-valued vectors, p(k) be a parameter vector of the current input and previous states and inputs*

$$[x(k-1), x(k-2)\ldots, u(k), u(k-1), \ldots,]$$

and $b_{i,j}$ be real numbers. For some integers K, n_K, a state-space model is described by

$$x(k) = \min_{i=1,\ldots,K} \max_{j=1,\ldots,n_i} \left(a_{i,j}^T p(k) + b_{i,j}\right). \tag{2-2}$$

This holds componentwise for vector-valued MMPS functions This form is called the conjunctive form.

This form is itself an MMPS system as described by Equation 2-1. The classes of Equation 2-1 and 2-2 coincide, meaning any MMPS system can be written in the form of 2-2 since min and max operations have the following properties: Let $\alpha, \beta, \gamma, \delta \in \mathbb{R}$

$$\min(\max(\alpha,\beta),\max(\gamma,\delta)) = \max(\min(\alpha,\gamma),\min(\alpha,\delta),\min(\beta,\gamma),\min(\beta,\delta)) \tag{2-3}$$

$$\max(\min(\alpha,\beta),\min(\gamma,\delta)) = \min(\max(\alpha,\gamma),\max(\alpha,\delta),\max(\beta,\gamma),\max(\beta,\delta)). \tag{2-4}$$

This is proven with lemma 28 from [17].

Another canonical form is the disjunctive form:

Definition 2.4. *Disjunctive canonical form: Let $\alpha_{i,j}$ be real-valued vectors, p(k) be a parameter vector of the current input and previous states and inputs*

$$[x(k-1), x(k-2)\ldots, u(k), u(k-1), \ldots,]$$

and $\beta_{i,j}$ be real numbers. For some integers K, m_K, a state-space model is described by

$$x(k) = \max_{i=1,\ldots,K} \min_{j=1,\ldots,m_i} \left(\alpha_{i,j}^T p(k) + \beta_{i,j}\right). \tag{2-5}$$

This holds componentwise for vector-valued MMPS functions. This form is called the disjunctive form.

Again this form is an MMPS function, and this form coincides with Equation 2-1. A different form is the Kripfganz form [19], using the difference between two MMPS functions.

Definition 2.5. *Kripfganz canonical form: let $\mu_{i,j}$ be real-valued vectors, p(k) be a parameter vector of the current input and previous states and inputs*

$$[x(k-1), x(k-2)\ldots, u(k), u(k-1), \ldots,]$$

and $\sigma_{i,j}$ be real numbers. For some integers K, L, a state-space model is described by:

$$\max_{i=1,\ldots L}(\mu_{1,i}p(k) + \sigma_{1,i}) - \max_{j=1,\ldots K}(\mu_{1,j}p(k) + \sigma_{1,j}) \tag{2-6}$$

This holds componentwise for vector-valued MMPS functions. This form is called the Kripfganz form.

This form also coincides with 2-1, with the property

$$\max(\alpha, \beta) = -\min(-\alpha, -\beta), \quad \min(\alpha, \beta) = -\max(-\alpha, -\beta) \tag{2-7}$$

which can be shown [17].

2-2 Piecewise affine functions

A convex polyhedron is a convex set that is the intersection of a finite number of half-spaces and is defined by its sides i.e. $Mx \leq Q$. Now let a polyhedral partition of polyhedron $\mathbb{D}$ be $\mathbb{D} = \bigcup_{i=0}^{\hat{N}} \Omega_i$ such that the finite amount of regions Ω_i do not overlap in the interior.

Definition 2.6. *A piecewise affine function (PWA function) $f : \mathbb{D} \to \mathbb{R}$ is a function that is affine in every region of the polyhedral partition $\mathbb{D}$. For example:*

$$\begin{aligned} f(x) &= \ell_{loc(i)}(x) \quad \forall x \in \Omega_i \\ \ell_{loc(i)} &= \alpha_k x + \beta_k. \end{aligned} \tag{2-8}$$

A PWA function is continuous if it is continuous on every boundary of its partition. Some local affine functions $\ell_{loc(i)}$ might occur in more than one subregion. Collect all unique affine functions and note the indices with $loc(i) \in \{1, 2, \ldots, M\}$

2-2-1 Equivalence with MMPS functions

Now if we partition these subregions Ω_i of $\mathbb{D}$ further into so called base regions $\mathbb{D}_{i,t}$ with $t \in \{0, 1, \ldots, m_i\}$. Now to make sure that no other affine function intersects with $\ell_{loc(i)}$ in the interior of $\mathbb{D})_{i,t}$, in other words:

$$\left\{x \mid \ell_j(x) = \ell_{\mathrm{loc}(i)}(x), j \neq \mathrm{loc}(i)\right\} \cap \mathrm{int}\,(\mathbb{D}_{i,t}) = \emptyset. \tag{2-9}$$

It is shown in [20] that all these regions can be partitioned such that $\Omega_i = \bigcup_{t=0}^{m_i} \mathbb{D}_{i,t}$ and:

1. The interior of each $\mathbb{D}_{i,t}$, $\text{int}(\mathbb{D}_{i,t})$ in short, is $\text{int}(\mathbb{D}_{i,t}) \neq \emptyset$
2. Let
$$\begin{aligned} I_{\geq,i,t} &= \left\{ j \mid \ell_j(x) \geq \ell_{\text{loc}(i)}(x), \forall x \in \mathbb{D}_{i,t} \right\}, \\ I_{\leq,i,t} &= \left\{ j \mid \ell_j(x) \leq \ell_{\text{loc}(i)}(x), \forall x \in \mathbb{D}_{i,t} \right\}, \end{aligned} \tag{2-10}$$
then for each $\mathbb{D}_{i,t}$:
$$I_{\geq,i,t} \cup I_{\leq,i,t} = \{1, 2, \ldots, M\}. \tag{2-11}$$
3. $\forall i, j \in \{1, 2, \ldots, \hat{N}\}, \bar{t} \in \{1, 2, \ldots, m_i\}, \hat{t} \in \{1, 2, \ldots, m_j\}, \bar{t} \neq \hat{t}, i \neq j$
$$\text{int}(\mathbb{D}_{i,\bar{t}}) \cap \text{int}(\mathbb{D}_{j,\hat{t}}) \neq \emptyset. \tag{2-12}$$

After this partition, renumber the regions $\mathbb{D}_{1,1}, \ldots, \mathbb{D}_{1,m_1}, \ldots, \mathbb{D}_{\hat{N},1}, \ldots, \mathbb{D}_{\hat{N},m_{\hat{N}}}$ to $\mathbb{D}_1, \ldots, \mathbb{D}_N$ with $N = m_1 + m_2 + \ldots + m_{\hat{N}}$ to simplify. Now define the active linear function as:

$$\ell_{\text{act}(i)} = \ell_{\text{loc}(j)}, \qquad \text{if } \mathbb{D}_i \subseteq \Omega_j \tag{2-13}$$

and determine the following index sets:

$$\begin{aligned} I_{\geq,i} &= \left\{ j \mid \ell_j(x) \geq \ell_{\text{act }(i)}(x), \quad \forall x \in \mathbb{D}_i \right\} \\ I_{\leq,i} &= \left\{ j \mid \ell_j(x) \leq \ell_{\text{act}(i)}(x), \quad \forall x \in \mathbb{D}_i \right\}. \end{aligned} \tag{2-14}$$

Then the PWA system in Equation 2-8 can be rewritten to an MMPS function in the disjunctive form as:

$$f(x) = \max_{i=1,\ldots,N} \left\{ \min_{j \in I_{\geq j}} \{\ell_j(x)\} \right\}, \quad \forall x \in \mathbb{D}, \tag{2-15}$$

or in the conjunctive form as:

$$f(x) = \min_{i=1,\ldots,N} \left\{ \max_{j \in I_{\leq j}} \{\ell_j(x)\} \right\}, \quad \forall x \in \mathbb{D}. \tag{2-16}$$

This means that any continuous piecewise affine function can also be written as an MMPS function. Though this process of dividing the regions and reconstructing them into an MMPS function is nontrivial. It also requires a complete description of all the regions. That is why an approximation of a PWA function can also be useful. Approximations can be obtained quite fast and if they are accurate enough they can substitute the need for an exact representation. They also do not require an exact description of all the regions of a PWA function.

2-3 Max-Plus Algebra

Next, we will discuss two distinct algebras that we will be using in specific neural network structures as will be explained in Section 4-1-2. The imposition of further constraints on the operations of MMPS functions gives rise to specific algebraic structures. Here we will discuss two of them, Max-Plus algebra and Min-Plus algebra. These algebraic structures have the advantage that some equations that are nonlinear in traditional algebra, are linear in these

specific algebras. Max-Plus algebra [21]–[24] consists of two main operations: addition and maximization. These operations are represented by the symbols $\oplus$ and $\otimes$:

$$x \oplus y = \max(x, y) \quad \text{and} \quad x \otimes y = x + y \tag{2-17}$$

for $x, y \in \mathbb{R}_\varepsilon \stackrel{\text{def}}{=} \mathbb{R} \cup \{-\infty\}$. The $\oplus$ operation is called *max-plus-algebraic addition* and the $\otimes$ operation is called *max-plus-algebraic multiplication*.

There are some similarities between the traditional $+$ and $\times$ operators, however, there are also some differences. One difference is the absence of inverse elements with respect to $\oplus$ in $\mathbb{R}_\varepsilon$, since this operator is *idempotent* i.e. $a \oplus a = a$ for $a \in \mathbb{R}_\varepsilon$.

The zero element for max-plus algebra is $\varepsilon := -\infty$. This has the properties

$$\begin{aligned} a \oplus \varepsilon &= \varepsilon \oplus a = a \\ a \otimes \varepsilon &= \varepsilon \otimes a = \varepsilon \quad \forall a \in \mathbb{R}_\varepsilon \end{aligned}$$

The identity element for max-plus algebra $\mathbb{1} := 0$, since $a \otimes \mathbb{1} = a \otimes 0 = a$. The structure of $(\mathbb{R}_\varepsilon, \otimes, \oplus)$ is then called max-plus algebra. The order of evaluation of the max-plus-algebraic operators is the same as the conventional plus-times algebra. This means that max-plus-algebraic multiplication has higher priority and max-plus-algebraic addition has lower priority.

2-3-1 Max-Plus matrix algebra

The max-plus-algebraic operations extend to matrix operations. Let $A, B \in \mathbb{R}_\varepsilon^{(m \times n)}$ and $C \in \mathbb{R}_\varepsilon^{(n \times p)}$, The $\oplus$ and $\otimes$ operators [21]–[24] for matrices are defined by:

$$\begin{aligned} (A \oplus B)_{ij} &= a_{ij} \oplus b_{ij} = \max\left(a_{ij}, b_{ij}\right) \\ (A \otimes C)_{ij} &= \bigoplus_{k=1}^{n} a_{ik} \otimes c_{kj} = \max_k \left(a_{ik} + c_{kj}\right). \end{aligned}$$

The $\bigoplus_{k=1}^{n}$ operator is analogous to the sum operator in conventional plus-times algebra.

Example 2.7. *Let* $A = \begin{bmatrix} 0 & \varepsilon & 2 \\ 2 & 3 & 4 \\ 3 & 2 & 3 \end{bmatrix}$ *and* $B = \begin{bmatrix} 2 & 1 & 2 \\ \varepsilon & 6 & 7 \\ 1 & 2 & \varepsilon \end{bmatrix}$.

Then $A \oplus B = \begin{bmatrix} 0 & \varepsilon & 2 \\ 2 & 3 & 4 \\ 3 & 2 & 3 \end{bmatrix} \oplus \begin{bmatrix} 2 & 1 & 2 \\ \varepsilon & 6 & 7 \\ 1 & 2 & \varepsilon \end{bmatrix} = \begin{bmatrix} 2 & 1 & 2 \\ 2 & 6 & 7 \\ 3 & 2 & 3 \end{bmatrix}$ *and*

$A \otimes B = \begin{bmatrix} 0 & \varepsilon & 2 \\ 2 & 3 & 4 \\ 3 & 2 & 3 \end{bmatrix} \otimes \begin{bmatrix} 2 & 1 & 2 \\ \varepsilon & 6 & 7 \\ 1 & 2 & \varepsilon \end{bmatrix} = \begin{bmatrix} 3 & 4 & 2 \\ 5 & 9 & 10 \\ 5 & 8 & 9 \end{bmatrix}$.

Note that the $\oplus$ operation stays communicative for matrices, but the $\otimes$ operator does not. The zero $(m \times n)$ matrix in max-plus algebra is $\mathcal{E}_{m \times n}$ which has ε for every element. The $(n \times n)$ max-plus identity matrix E_n which has 0s on the diagonal and ε everywhere else.

2-4 Min-Plus Algebra

Min-plus algebra is much analogous to max-plus algebra, the definition for min-plus algebra, similar to Equation 2-17, is:

$$x \oplus' y = \min(x, y) \quad \text{and} \quad x \otimes' y = x + y. \tag{2-18}$$

The zero element for min-plus algebra is $\top := \infty$, such that

$$\begin{aligned} a \oplus' \top &= \top \oplus' a = a \\ a \otimes' \top &= \top \otimes' a = \top \quad \forall a \in \mathbb{R}_\top := \mathbb{R} \cup \{\top\}. \end{aligned}$$

The tuple $(\mathbb{R}_\top, \otimes', \oplus')$ is called the min-plus algebra. Also similar are matrix multiplications. Let $A, B \in \mathbb{R}_\top^{(m \times n)}$ and $C \in \mathbb{R}_\top^{(n \times p)}$, The $\oplus'$ and $\otimes'$ operators for matrices are defined by:

$$\begin{aligned} (A \oplus' B)_{ij} &= a_{ij} \oplus' b_{ij} = \min\left(a_{ij}, b_{ij}\right) \\ (A \otimes' C)_{ij} &= \bigoplus_{k=1}^{n\ \prime} a_{ik} \otimes' c_{kj} = \min_k \left(a_{ik} + c_{kj}\right) \end{aligned}$$

Note that for a scalar case, the $\otimes$ operator is the same as the $\otimes'$ operator, but not for matrix multiplication.

Chapter 3

Model Predictive Control

Model predictive control (MPC) is an optimal control method that uses a model to make predictions about the future behavior of the system. It can also handle constraints on the states and inputs. This chapter discusses linear MPC controllers and how to formulate the MPC problem as a quadratic programming problem. Then it discusses stability for linear MPC controllers and expands this to nonlinear controllers. Next, we discuss Mixed Logical Dynamical (MLD) and how to use MPC on these nonlinear systems. We also briefly touch on explicit MPC.

3-1 Optimal Control: Linear Quadratic Regulator

Consider a linear time-invariant system

$$\begin{aligned} x(k+1) &= Ax(k) + Bu(k) \\ y(k) &= Cx(k) + Du(k) \\ x_0 &= x(0) \end{aligned} \tag{3-1}$$

with $x \in \mathbb{R}^n, u \in \mathbb{R}^m$ and $y \in \mathbb{R}^p$, and the matrices $A \in \mathbb{R}^{n \times n}, B \in \mathbb{R}^{n \times m}, C \in \mathbb{R}^{p \times n}$ and $D \in \mathbb{R}^{p \times n}$. With this information on how the state evolves, the states $x(k)$ can be rewritten to:

$$\begin{aligned} x(k) &= A^k x_0 + \sum_{j=1}^{k} A^{k-j-1} u(j) \\ &= A^k x_0 + \underbrace{\begin{bmatrix} B & AB & \dots & A^{k-1}B \end{bmatrix}}_{=\mathcal{C}_k} \underbrace{\begin{bmatrix} u(k-1) \\ u(k-2) \\ \vdots \\ u(0) \end{bmatrix}}_{=u(0:k-1)} \\ &= A^k x_0 + \mathcal{C}_k u(0:k-1). \end{aligned} \tag{3-2}$$

Observe that the states $x(k)$ are a function of the previous inputs $u(1:k)$ and the initial state $x(0)$. Now the goal is to find a controller that balances the states and outputs. Consider the cost function V with prediction horizon N:

$$V(x(0), u(0:N-1)) = \frac{1}{2}\sum_{k=0}^{N-1}\left[x(k)^T Q x(k) + u(k)^T R u(k)\right] + \frac{1}{2}x(N)^T P_f x(N). \tag{3-3}$$

In this cost function, there are 3 relevant matrices: the Q matrix putting a weight on the states $x(0:N-1)$, an R matrix putting a weight on the inputs $u(0:N-1)$, and a matrix P_f on the terminal state $x(N)$. Having larger entries in the Q matrix compared to the R matrix means the goal is to drive the states to zero, with lesser regard to large control inputs. Having larger R matrix entries means the states might be slower to be driven to zero, but the controller effort is also lower. Choosing these matrices requires some tuning and is not too straightforward. The matrices Q, R, and P_f need to be real and symmetric, the matrix Q and P_f matrix need to be positive semi-definite and the R matrix needs to be positive definite. To obtain a controller that finds the optimal control inputs u the cost function needs to be optimized:

$$\begin{aligned} &\underset{u(1:N-1)}{\text{minimize}} \quad V\left(x(0), u(0:N-1)\right) \\ &\text{s.t.} \quad x(k+1) = Ax(k) + Bu(k) \quad \text{for } k = 0, 1, \ldots, N-1. \end{aligned} \tag{3-4}$$

If there are no constraints in the optimization problem in 3-4, the result of this Linear Quadratic Regulator (LQR) problem is famously known from [25] to be the control law $u(k) = K(k)x(k)$ with gain $K(k)$ computed with:

$$\begin{aligned} P(k-1) = Q + A^T P(k) A - \left(A^T P(k) B\right)\left(B^T P(k) B + R\right)^{-1}\left(B^T P(k) A\right) \\ \forall k = N, N-1, \ldots, 1 \\ P(N) = P_f \end{aligned} \tag{3-5}$$

also known as the backward Ricatti iteration. This yields the optimal gain

$$K(k) = -\left(B^T P(k+1) B + R\right)^{-1} B^T P(k+1) A \quad \forall k = N-1, N-2, \ldots, 0. \tag{3-6}$$

For an infinite horizon, this gain becomes constant, and the control law becomes $u(k) = Kx(k)$ with

$$\begin{aligned} P = Q + A^T P A - \left(A^T P B\right)\left(R + B^T P B\right)^{-1}\left(B^T P A\right), \\ K = -(B^T P B + R)^{-1}(B^T P A) \end{aligned} \tag{3-7}$$

This does not yet consider any constraints on the inputs or states. Section 3-3-1 will discuss this situation further.

3-1-1 Receding horizon principle

In MPC [1], [2], the first input of the sequence at time t $u(t+0), u(t+1), \ldots u(t+N)$ with N as the prediction horizon is actually applied. After this the time window shifts to $t+1$ and the optimization problem in 3-4 is recomputed for a new x_0 to obtain a new sequence $u(t+0), u(t+1), \ldots, u(t+N)$. This is called a receding horizon control approach. Often with this technique, there is also a control horizon N_c, after which the control is kept constant to reduce the computational complexity of the optimization problem. This is illustrated in Figure 3-1 .

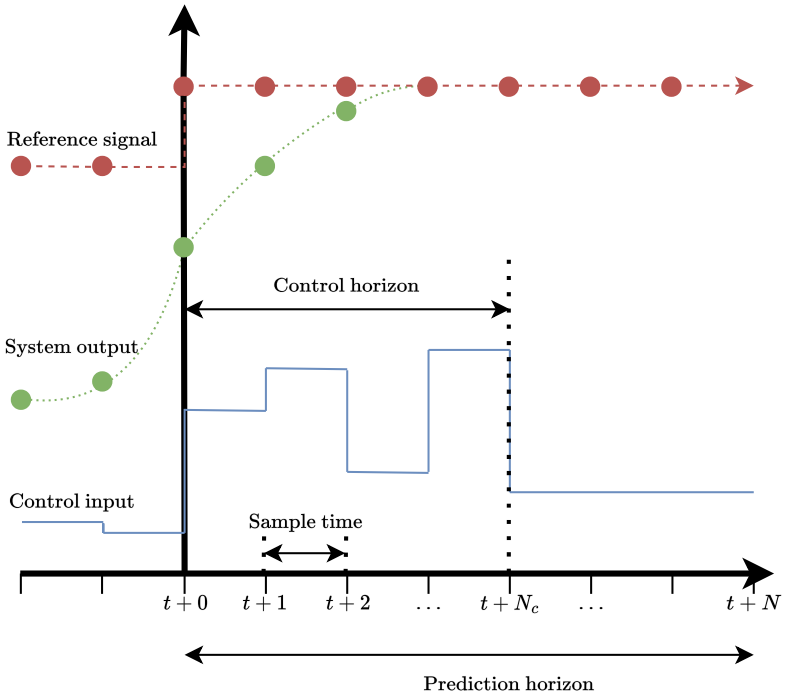

Figure 3-1: Principles of mpc

3-2 Quadratic programming

To solve the MPC problem of a linear system with n states and m inputs with a quadratic cost function, we can reformulate the problem into a quadratic programming problem. A general form for quadratic programming is:

$$\min_z \frac{1}{2} z^T H z + c^T z \text{ s.t. } Jz \leq m \tag{3-8}$$

By rewriting in this form we can use an efficient algorithm such as the dual method described in [26] to solve this problem. Now to rewrite the optimization problem we first define the states and inputs as stacked vectors

$$\bar{x} = \begin{bmatrix} x(k+1) \\ x(k+2) \\ \vdots \\ x(k+N) \end{bmatrix}, \bar{u} = \begin{bmatrix} u(k) \\ x(k+1) \\ \vdots \\ x(k+N-1) \end{bmatrix} \tag{3-9}$$

and modify the state and cost matrices as follows

$$\begin{aligned}\bar{A} &= \begin{bmatrix} A \\ \vdots \\ A^N \end{bmatrix} \in \mathbb{R}^{Nn\times n}, \bar{B} \begin{bmatrix} B & 0 & \cdots & 0 \\ \vdots & \ddots & \ddots & \vdots \\ \vdots & & \ddots & 0 \\ A^{N-1}B & \cdots & \cdots & B \end{bmatrix} \in \mathbb{R}^{Nn\times Mn}, \\ \bar{Q} &= \begin{bmatrix} Q \\ Q \\ \vdots \\ P \end{bmatrix} \in \mathbb{R}^{Nn\times n}, \bar{R} = \begin{bmatrix} R \\ R \\ \vdots \\ R \end{bmatrix} \in \mathbb{R}^{Nm\times m}\end{aligned} \tag{3-10}$$

to get the equation

$$\bar{x} = \bar{A}x_0 + \bar{B}\bar{u}. \tag{3-11}$$

The cost function now becomes

$$V(x_0, \bar{u}) = \bar{x}\bar{Q}\bar{x} + \bar{u}\bar{R}\bar{u} \tag{3-12}$$

And we can rewrite this in quadratic form as follows:

$$\begin{aligned} V(x_0, \bar{u}) &= \bar{x}\bar{Q}\bar{x} + \bar{u}\bar{R}\bar{u} \\ &= \left(\bar{A}x_0 + \bar{B}\bar{u}\right)^T \bar{Q}\left(\bar{A}x_0 + \bar{B}\bar{u}\right) + \bar{u}\bar{R}\bar{u} \\ &= \left(x_0^T\bar{A}^T + \bar{u}^T\bar{B}^T\right)\bar{Q}\left(\bar{A}x_0 + \bar{B}\bar{u}\right) + \bar{u}\bar{R}\bar{u} \\ &= \underbrace{x_0^T\bar{A}^T\bar{Q}\bar{A}x_0}_{\text{does not depend on } \bar{u}} + 2x_0^T\bar{A}^T\bar{Q}\bar{B}\bar{u} + \bar{u}^T\bar{B}^T\bar{Q}\bar{B}\bar{u} + \bar{u}\bar{R}\bar{u}. \end{aligned} \tag{3-13}$$

Since there is a part of the equation that does not depend on u, we can leave it out of the optimization. We get a new function V_{opt}:

$$\begin{aligned} V_{\text{opt}}(x_0, \bar{u}) &= \bar{u}^T \underbrace{\left(\bar{B}^T\bar{Q}\bar{B} + R\right)}_{H}\bar{u} + \underbrace{2x_0^T\bar{A}^T\bar{Q}\bar{B}}_{c^T}\bar{u} \\ &= \bar{u}^T H\bar{u} + c^T\bar{u} \end{aligned} \tag{3-14}$$

which is in the form of 3-8. Now if all constraints are linear inequality constraints we can use $\bar{x}$ and $\bar{u}$ to construct the constraint matrix J and constraint vector m to the same form as 3-8.

3-3 Stability concepts

To determine if a model predictive controller is stable, we first need a proper definition of stability. First start with some definitions for invariant sets:

Definition 3.1. *Positive invariance: Suppose a dynamical system is $x(k+1) = f(x(k))$ and a trajectory is $x(k, x_0)$ with initial point x_0. A set $P = \{x \in \mathbb{R}^n | \psi(x) \leq 0\}$ where ψ is a real-valued function is said to be positive invariant if $x_0 \in P$ implies that $x(t, x_0) \in P \quad \forall t \geq 0$. That is to say, any trajectory of the system that enters P stays in P forever.*

Definition 3.2. *Control invariance: A set P is control invariant for $x(k+1) = f(x(k), u(k))$ and $u(k) = \kappa(x(k))x(k)$ with $u \in \mathbb{U}$ if for all $x \in P$ there exists a $u \in \mathbb{U}$ that follows such that $f(x, u) \in P$. In other words, for a dynamic system, there exists a set for which the trajectory stays in the set with only one control input following the control law.*

Next, we need definitions for different classes of functions for a definition of stability:

Definition 3.3. *Class $\mathcal{K}, \mathcal{K}_\infty$ and $\mathcal{KL}$ functions: A function α: $[0, a] \to [0, \infty]$ is a class $\mathcal{K}$ function if it is:*

- *Strictly increasing*
- $\alpha(0) = 0$

A function α is class $\mathcal{K}_\infty$ if it is:

- *class $\mathcal{K}$*
- $a = \infty$
- $\lim_{r\to\infty} \alpha(r) = \infty$

A continuous function β: $[0, a) \times [0, \infty) \to [0, \infty)$ is said to be class $\mathcal{KL}$ if'

- *for each fixed s $\beta(r, s)$ belongs to class $\mathcal{K}$*
- *for each fixed r, the function $\beta(r, s)$ is decreasing with respect to s, so $\beta(r, s) \to 0$ for $s \to \infty$*

With these definitions, we can construct a definition for asymptotical stability. First denote the state of a dynamical system at timestep k with initial condition x_0 and control sequence $\boldsymbol{u} := u(0 : k)$ by $\phi(k, x_0, \boldsymbol{u})$.

Definition 3.4. *(Global) Asymptotical stability: Suppose $\mathbb{X}$ is positive invariant for $x(k+1) = f(x(k))$. The origin of the system is asymptotically stable for $x(k+1) = f(x(k))$ in $\mathbb{X}$ if there exist a class $\mathcal{KL}$ function β such that for each $x \in \mathbb{X}$*

$$|\phi(s; x)| \leq \beta(|x|, s) \quad \forall s \in \mathbb{S}_{\geq 0}. \tag{3-15}$$

The origin of the system is globally asymptotically stable if $\mathbb{X} = \mathbb{R}^n$.

Now we have a definition of stability, but to prove that a system is stable we need some more definitions. One way to prove the stability of a system is by using Lyapunov functions. They are defined as the following:

Definition 3.5. *Lyapunov function: suppose that $\mathbb{X}$ is positive invariant for $x(k + 1) = f(x(k))$. A function $V : \mathbb{R}^n \to \mathbb{R}_{\geq 0}$ is said to be a Lyapunov function in $\mathbb{X}$ for $x(k + 1) = f(x(k))$ if there exists functions $\alpha_1, \alpha_2 \in \mathcal{K}_\infty$ and a continuous positive definite function α_3 such that for any $x \in \mathbb{X}$*

$$V(x) \geq \alpha_1(|x|) \tag{3-16}$$

$$V(x) \leq \alpha_2(|x|) \tag{3-17}$$

$$V(f(x)) - V(x) \leq -\alpha_3(|x|). \tag{3-18}$$

To prove stability for a system we can now use the Lyapunov stability theorem:

Theorem 3.6. *Lyapunov stability theorem (Theorem B.24 in [27]): Suppose $\mathbb{X} \in \mathbb{R}^n$ is positive invariant for $x(k+1) = f(x(k))$. If a Lyapunov function exists in $\mathbb{X}$ for the system, then the origin is asymptotically stable in $\mathbb{X}$ for $x(k+1) = f(x(k)$. The origin of the system is globally asymptotically stable if $\mathbb{X} = \mathbb{R}^n$. If $\alpha_j(|x|) = c_j|x|^a$ with $a, c_i \in \mathbb{R}_{\geq 0}, i = 1, 2, 3$, the origin of the system is exponentially stable.*

This theorem is proven by [27] in Theorem B.24.

3-3-1 Stability for constrained linear systems

Using the previous theorem, we can determine stability for a linear system with linear state, and control input constraints. These constraints do not allow us to simply use the control law from the linear quadratic regulator. Consider the linear time-invariant system in Equation 3-1. with the linear state and control input constraints as

$$x \in \mathcal{X} \quad u \in \mathcal{U} \tag{3-19}$$

Now consider a general cost function with stage cost V_s and terminal cost V_f

$$V(x(0), u(0:N-1)) = V_s(x(0), u(0:N-1)) + V_f(x(N)) \tag{3-20}$$

An example is a quadratic cost function, the same as the linear quadratic regulator in Equation 3-3:

$$V(x(0), u(0:N-1)) = \frac{1}{2}\sum_{k=0}^{N-1}\left[x(k)^T Q x(k) + u(k)^T R u(k)\right] + \frac{1}{2}x(N)^T P_f x(N). \tag{3-21}$$

The optimization problem becomes:

$$\begin{aligned} \underset{u(0:N-1)}{\text{minimize}} \quad & V\left(x(0), u(0:N-1)\right) \\ \text{s.t.} \quad & x(k+1) = Ax(k) + Bu(k) \\ & x(k) \in \mathcal{X} \quad \forall k = 0, 1, \ldots, N-1 \\ & u(k) \in \mathcal{U} \quad \forall k = 0, 1, \ldots, N-1 \\ & x(N) \in \mathcal{X}_f \end{aligned} \tag{3-22}$$

Here $\mathcal{X}_f$ is a terminal region to which we want the controller to drive the states. This could be the origin. It can also be a set that contains the origin. The solution of the optimization problem yields a control sequence $u^*(0:N-1)$ and using the receding horizon principle only $u^*(0)$ is applied to the system. The closed loop system is then $x(t+1) = Ax(t) + Bu^*(x(t))$, where u^* is only a function of the state, conserving time-invariance.

Now that there are constraints on the system, the optimization problem is not feasible for all $x(0)$, as the controller effort required to end up in $\mathcal{X}_f$ could violate the constraints. The set of all possible $x(0)$ where a solution to the optimization problem exists is $\mathcal{X}_0$.

Now consider the following theorem as Theorem 12.2 from [28]:

Theorem 3.7. *Consider a model predictive controller for a linear system with constraints as in Equation 3-19. Now assume that*

(A) The stage cost $V_s(x(0), u(0:N-1))$ and terminal cost $V_f(x(N))$ are continuous and positive definite functions.

(B) The sets $\mathcal{U}, \mathcal{X}$, and $\mathcal{X}_f$ are compact and have the origin in their interior.

(C) $\mathcal{X}_f$ is control invariant and $\mathcal{X}_f \in \mathcal{X}$.

(D) $\min_{v \in \mathcal{U},\, Ax+Bv \in \mathcal{X}_f} (-V_f(x) + V_s(x,v)) + V_f(Ax+Bv)) \leq 0, \forall x \in \mathcal{X}_f$. Where the functions $V_f(x)$ and $V_f(x,v)$ are the terminal and stage cost functions.

Then the origin of the closed-loop system is asymptotically stable with the domain of attraction $\mathcal{X}_0$.

This theorem is proven in chapter 12 of [28], it uses the cost function $V(\cdot)$ as a Lyapunov function to prove stability using Theorem 3.6.

The cost function defined in 3-21 satisfies (A), the defined constraint sets $\mathcal{U}, \mathcal{X}$, and $\mathcal{X}_f$ are assumed to be compact and have the origin in their interior, satisfying (B). Now all there is left to check if conditions (C) and (D) are satisfied. With these conditions fulfilled, any feasible solutions to the optimization problem prove stability a posteriori, and by choosing different $x(0)$, the domain of attraction $\mathcal{X}_0$ can be determined with algorithm 10.3 from [28].

3-3-2 Stability for nonlinear systems

Now consider a constrained nonlinear system, where $\mathbb{X}$ is the set of non-constrained $x(t)$, $\mathbb{U}$ the set of non-constrained $u(t)$ and $\mathbb{Z}$ the set of non-constrained $x(t)$ and $u(t)$ combined. $\kappa(x(t))$ is the control law that arises from the MPC controller.

$$\begin{aligned} x(t+1) &= f(x(t), u(t)) \\ u(t) &= \kappa(x(t)) \end{aligned} \tag{3-23}$$

This control law arises from solving the following optimization problem and applying the first control input $u(0)$ that is found.

$$\begin{aligned} \underset{u(0:N-1)}{\text{minimize}} \quad & V\left(x(0), u(0:N-1)\right) \\ \text{s.t.} \quad & x(t+1) = f(x(t), u(t)) \quad \forall t = 0, 1, \ldots, N-1 \\ & x(t) \in \mathcal{X} \quad \forall t = 0, 1, \ldots, N-1 \\ & u(t) \in \mathcal{U} \quad \forall t = 0, 1, \ldots, N-1 \\ & u(t), x(t) \in \mathcal{Z} \subseteq \mathcal{X} \times \mathcal{U} \quad \forall t = 0, 1, \ldots, N-1 \\ & x(N) \in \mathcal{X}_f \\ & \mathcal{U} \subseteq \mathbb{U}, \quad \mathcal{X} \subseteq \mathbb{X}, \quad \mathcal{Z} \subseteq \mathbb{Z} \end{aligned} \tag{3-24}$$

With the cost function in Equation 3-20. Now to prove stability of the equilibrium for a nonlinear as Equation 3-23, we can modify the assumptions of Theorem 3.7.By assuming the

set $\mathcal{X}_f$ is control invariant we satisfy assumption C, and by modifying the terminal cost to be $V_f(f(x,u))$ instead of $V_f(Ax+Bv)$ in condition D and confirming that it is indeed a control Lyapunov function, we also satisfy this condition. This gives us enough to prove asymptotic stability of the origin.

3-4 MPC for PWA systems

Consider a system with the state transitions determined by a piecewise affine function. A PWA system with s regions would be in the form:

$$x(k) = \begin{cases} A_1x(k) + B_1u(k) + F_1 \text{ if } x(k), u(k) \in \Omega_1 \\ A_2x(k) + B_2u(k) + F_2 \text{ if } x(k), u(k) \in \Omega_2 \\ \vdots \\ A_sx(k) + B_su(k) + F_s \text{ if } x(k), u(k) \in \Omega_s \end{cases} \tag{3-25}$$

where each Ω_i is a linear region. We assume the linear regions are polytopic and can therefore be defined by a set of linear inequalities.

3-4-1 Mixed Logical Dynamical model

To control a PWA system, we have to write it in a different form, namely as a Mixed-Logical-Dynamical (MLD) [13] model. This model has the shape:

$$\begin{aligned} x(k+1) &= Ax(k) + B_1u(k) + B_2\delta(k) + B_3z(k) \\ E_1x(k) + E_2u(k) + E_3\delta(k) + E_4z(k) &\leqslant g_5 \end{aligned} \tag{3-26}$$

where $\delta(k)$ is a vector of binary variables ($\delta_i(k) \in \{0,1\}$) and $z(k)$ a vector of auxiliary variables defined by $z_i(k) = \delta_i(k)x(k)$ or $z_i(k) = \delta_i(k)u(k)$.

We can introduce the relation $[\delta_i(k) = 1 \leftrightarrow x(k) \in \Omega_i]$ and $[\delta_i(k) = 0 \leftrightarrow x(k) \notin \Omega_i]$. For a simple system we assume that all inputs $u(k)$ are allowed, such that every Ω_i does not depend on $u(k)$, but only on $x(k)$. Then the piecewise affine system becomes

$$x[k+1] = \begin{cases} A_1x(k) + B_1u(k) + F_1 \text{ if } \delta_1(k) = 1 \\ A_2x(k) + B_2u(k) + F_2 \text{ if } \delta_2(k) = 1 \\ \vdots \\ A_sx(k) + B_su(k) + F_s \text{ if } \delta_s(k) = 1. \end{cases} \tag{3-27}$$

From this formulation we can derive that the sum of these binary variables $\delta_i(k)$ is always equal to 1, so

$$\sum_{i=1}^{s} \delta_i(k) = 1. \tag{3-28}$$

To rewrite the constraint 3-28 we have column vector $\mathbf{1}$ filled with ones, such that we have the constraints:

$$\begin{aligned} \mathbf{1}\delta_i(k) &\leq 1 \\ -\mathbf{1}\delta_i(k) &\leq -1. \end{aligned} \tag{3-29}$$

Our piecewise affine system in 3-25 is assumed to be well posed, which means that

$$\begin{aligned} &\Omega_i \cap \Omega_j = \emptyset \quad \forall i \neq j \\ &\bigcup_{i=1}^{s} \Omega_i = \Omega. \end{aligned} \tag{3-30}$$

We can define these polytopic linear regions with a set of inequalities

$$\Omega_i : \left\{ \begin{bmatrix} x \\ u \end{bmatrix} : S_i x + R_i u \leq T_i \right\}. \tag{3-31}$$

By contradiction, we can write this as the inequality:

$$S_i x(k) + R_i u(k) - T_i \leq M_i^* \left(1 - \delta_i(k)\right) \tag{3-32}$$

with $M_i^* := \max_{x \in \Omega} S_i x(k) + R_i u(k) - T_i$.

Now Equation 3-27 can be rewritten to

$$x(k+1) = \sum_{i=1}^{s} \left(A_i x(k) + B_i u(k) + F_i\right) \delta_i(k). \tag{3-33}$$

Though we can simplify by using an auxiliary variable:

$$x(k+1) = \sum_{i=1}^{s} z_i(k) \tag{3-34}$$

with $z_i(k) := \left(A_i x(k) + B_i u(k) + F_i\right) \delta_i(k)$. This can be ensured with the following inequalities:

$$\begin{aligned} &z_i(k) \leq M \delta_i(k) \\ &z_i(k) \geq m \delta_i(k) \\ &z_i(k) \leq A_{di} x(k) + B_{di} u(k) + B_{1di} - m\left(1 - \delta_i(k)\right) \\ &z_i(k) \geq A_{di} x(k) + B_{di} u(k) + B_{1di} - M\left(1 - \delta_i(k)\right). \end{aligned} \tag{3-35}$$

with m and M defined as:

$$\begin{aligned} &M = [M_1, M_2, \dots M_n]^T, \\ &M_j := \max_{i=1,\dots,s} \left\{ \max_{x \in \Omega} A_i^j x + B_i^j u \right\}, \\ &m = [m_1, m_2, \dots m_n]^T, \\ &m_j := \min_{i=1,\dots,s} \left\{ \max_{x \in \Omega} A_i^j x + B_i^j u \right\}. \end{aligned} \tag{3-36}$$

Where A_i^j denotes the jth row of A_i. These can be computed by solving 2ns linear programs, or estimated.

Now with the constraints of Equations 3-29, 3-32 and 3-35 and the relation 3-34, the system can be rewritten to a simplified form of 3-26:

$$\begin{aligned} &x(k+1) = B_3 z(k) \\ &E_1 x(k) + E_2 u(k) + E_3 \delta(k) + E_4 z(k) \leqslant g_5 \end{aligned} \tag{3-37}$$

3-4-2 Optimal control of MLD system

We can control an MLD system using MPC by a procedure given in [28]. Consider the following cost function:

$$V_0\left(x(0), u_{0:N-1}\right) = \|Px(N)\|_p + \sum_{k=0}^{N-1} \|Qx(k)\|_p + \|Ru(k)\|_p + \|Q_\delta \delta(k)\|_p + \|Q_z z(k)\|_p \quad \text{(3-38)}$$

Where the terms are evaluated with a p-norm. For this section, we will consider only the 1-norm and the ∞-norm as they simplify the problem to a mixed integer linear programming problem.

Simplify by setting Q_δ and Q_z to zero gives us the following cost function:

$$V_0\left(x(0), u(0:N-1)\right) = \|Px(N)\|_p + \sum_{k=0}^{N-1} \|Qx(k)\|_p + \|Ru(k)\|_p \quad \text{(3-39)}$$

that is subject to the constraints:

$$\begin{aligned} x(k+1) &= Ax(k) + B_1 u(k) + B_2 \delta(k) + B_3 z(k) \quad k = 0, 1...N-1 \\ E_1 x(k) &+ E_2 u(k) + E_3 \delta(k) + E_4 z(k) \leqslant g_5 \quad k = 0, 1...N-1 \end{aligned} \quad \text{(3-40)}$$

The optimal control problem is:

$$\begin{aligned} V^*(x(0)) = &\min_{u_{0:N-1}} V_0\left(x(0), u_{0:N-1}\right) \\ \text{s.t.} \quad & x(N) \in \mathcal{X}_f \\ & x(k+1) = Ax(k) + B_1 u(k) + B_2 \delta(k) + B_3 z(k) \quad k = 0, 1...N-1 \\ & E_1 x(k) + E_2 u(k) + E_3 \delta(k) + E_4 z(k) \leqslant g_5 \quad k = 0, 1...N-1 \end{aligned} \quad \text{(3-41)}$$

Now the goal is to find a suitable form for the function V. The choice of 1-norm or ∞ is motivated by the following theorem, which is proven in Theorem 17.3 in [28]:

Theorem 3.8. *Consider an optimal control problem in the form of 3-41 with cost function 3-39 and the system is well-posed. Then there exists a solution in the form of a polyhedral PWA state feedback control law.*

In theory, there exsists a polyhedral PWA state feedback control law for a well-posed system.

From the standard form in Equation 3-26, we can derive an expression for each $x(k)$:

$$x(k) = A^k x(0) + \sum_{j=0}^{k-1} A^j \left(B_1 u(k-1-j) + B_2 \delta(k-1-j) + B_3 z(k-1-j)\right) \quad \text{(3-42)}$$

With our system from 3-37 this simplifies to:

$$x(k) = B_3 z(k-1). \quad \text{(3-43)}$$

Now consider the vector $\varepsilon = \{\varepsilon_0^u, \varepsilon_1^u, \ldots, \varepsilon_{N-1}^u, \varepsilon_0^x, \varepsilon_1^x, \ldots, \varepsilon_N^x\}$ that are subject to:

$$\begin{aligned}
-\mathbf{1}_m\varepsilon_k^u &\leq Ru(k), \quad k = 0, 1, \ldots, N-1 \\
-\mathbf{1}_m\varepsilon_k^u &\leq -Ru(k), \quad k = 0, 1, \ldots, N-1 \\
-\mathbf{1}_n\varepsilon_k^x &\leq Qx(k), \quad k = 0, 1, \ldots, N-1 \\
-\mathbf{1}_n\varepsilon_k^x &\leq -Qx(k), \quad k = 0, 1, \ldots, N-1 \\
-\mathbf{1}_n\varepsilon_N^x &\leq Px(N) \\
-\mathbf{1}_n\varepsilon_N^x &\leq -Px(N)
\end{aligned} \tag{3-44}$$

which changes the cost function of Equation 3-39 to a sum of this vector, now we obtain:

$$V(\varepsilon) = \varepsilon_0^u + \varepsilon_1^u + \ldots + \varepsilon_{N-1}^u + \varepsilon_0^x + \varepsilon_1^x + \ldots + \varepsilon_N^x \tag{3-45}$$

So now we have the optimization problem:

$$\begin{aligned}
&\min_{\varepsilon} V(\varepsilon) \\
\text{subj. to } &-\mathbf{1}_m\varepsilon_k^u \leq \pm Ru(k) \quad k = 0, 1, \ldots, N-1 \\
&-\mathbf{1}_n\varepsilon_k^x \leq \pm QB_3z(k-1) \quad k = 1, \ldots, N-1 \\
&-\mathbf{1}_n\varepsilon_N^x \leq \pm PB_3z(N-1) \\
&x(k+1) = B_3z(k) \quad k = 0, 1, \ldots, N-1 \\
E_1x(k) + E_2u(k) + E_3\delta(k) + E_4z(k) &\leqslant g_5 \quad k = 0, 1, \ldots, N-1 \\
&x(N) \in \mathcal{X}_f
\end{aligned} \tag{3-46}$$

3-5 Explicit MPC for linear systems

A drawback of model predictive control is that an optimization problem has to be solved at every timestep. If the optimization problem becomes too complex, it could take longer than the sampling time, causing problems as well, though optimization algorithms and hardware have been getting faster over the years. Another disadvantage is that some optimizers require licenses to use. A solution that would not require an optimization problem at each step is explicit model predictive control [3]. This method uses multiparametric programming techniques to reduce the control law to a simple to evaluate function.

In explicit model predictive control, an explicit control law is computed offline, and using this relatively fast-to-execute control law the control inputs are computed. With some conditions, such as a quadratic cost function and linear constraints, [29] shows that the resulting closed-form solution is continuous and piecewise linear. A control law could then be

$$u(t) = \begin{cases} F_1x + g_1 & x(t) \in \Gamma_1 \\ F_2x + g_2 & x(t) \in \Gamma_2 \\ \quad \vdots & \\ F_fx + g_f & x(t) \in \Gamma_f \end{cases} \tag{3-47}$$

Where $\Gamma_1, \Gamma_2, \ldots, \Gamma_f$ are convex polyhedral regions. Explicit control solutions such as in Equation 3-47 can be found by solving multiparametric linear programs.

3-5-1 Memory use

The control law $\kappa(x(t))$ as the result of a linear MPC controller is known to be continuous and piecewise-affine [29] as defined by Definition 2.6. For a linear system with $A \in \mathbb{R}^{n_x}$ and $B \in \mathbb{R}^{n_u}$, the memory use is

$$\Gamma_\kappa = \alpha_{\text{bit}} \left(n_h(n_x + 1) + n_f(n_x n_u + n_u) \right) \tag{3-48}$$

with n_h the number of unique hyperplanes, n_f the number of unique feedback laws and α_{bit} the size of one number in memory. The main disadvantage of using explicit MPC is that the number of linear regions increases with the degrees of freedom of the optimization problem. This is influenced by the prediction horizon, the number of controls, and the number of constraints with an exponential relation. It also increases with the number of states, until the number of states is larger than the number of constraints [29]. A large number of linear regions increases memory use.

Chapter 4

Neural Networks

Neural networks, also known as artificial neural networks, are inspired by the name and structure found in brains. They are used as a tool for regression and can approximate complicated functions. This chapter will explain what neural networks consist of and what their connections are to max-min-plus-scaling functions. It also gives a measure of complexity for a neural network that is comparable to the complexity of max-min-plus-scaling functions. It then explains how neural networks are trained to get a good approximation of a function.

4-1 Neural Networks

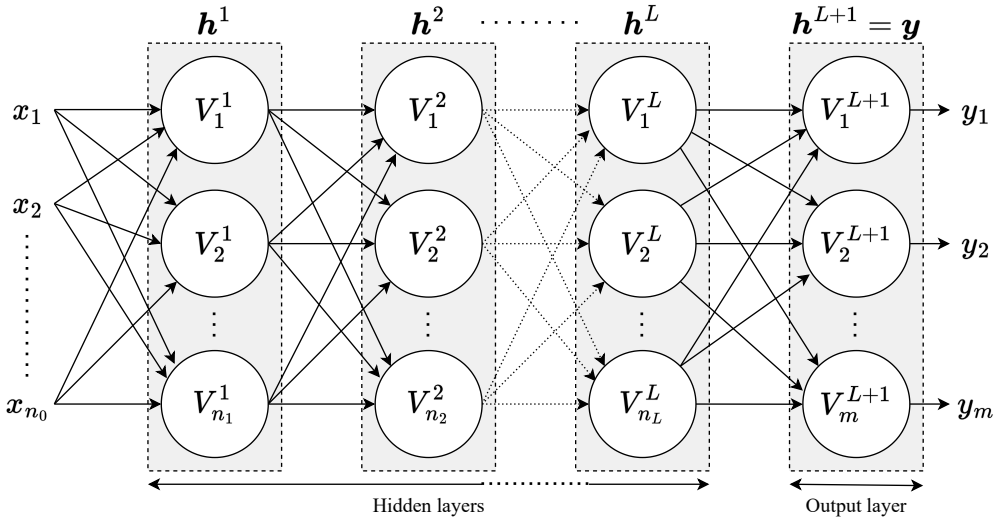

Figure 4-1: Description of a neural network

A neural network has a set of n_0 inputs in a vector $\boldsymbol{x} = [x_1, x_2, \ldots, x_{n_0}]^T$ and a set of m outputs in a vector $\boldsymbol{y} = [y_1, y_2, \ldots, y_m]^T$. A neural network has nodes structured in layers, let

$l \in 1, 2, \ldots, L$ with L the number of hidden layers containing $n_1, n_2 \ldots n_L$ nodes respectively. Also, it contains an output layer $L+1$, which is often slightly different than the hidden layers. The inputs of each hidden layer are denoted as $\boldsymbol{z}^l = \{z_1^l, z_2^l, \ldots, z_{n_l}^l\}$. The outputs of each hidden layer, meaning the result of an activation function [30], of the hidden layers are given by $\boldsymbol{h}^l = \{h_1^l, h_2^l, \ldots h_{n_l}^l\}$. The output layer is not hidden and often has an activation function that is different from the hidden layers. Each layer has a weight matrix W^l of $n_l \times n_{l-1}$ and bias vector $\boldsymbol{b}^l$ of $n_l \times 1$ to obtain the activation values. The weights are the coefficients of the equation we are trying to resolve, and multiply the inputs of the layer before we apply the activation functions. The biases make sure that when the input is zero, there is still a nonzero output, as varying the weights would not change this. Given activation functions $\boldsymbol{f}^l(x)$ and $f^{L+1}(x)$, the activations and outputs are given by:

$$\begin{aligned} \boldsymbol{z}^1 &= W^1 \boldsymbol{x} + \boldsymbol{b} \\ \boldsymbol{h}^1 &= \boldsymbol{f}^1\left(\boldsymbol{z}^1\right) \\ \boldsymbol{z}^l &= W^l \boldsymbol{h}^{l-1} + \boldsymbol{b}^l \\ \boldsymbol{h}^l &= \boldsymbol{f}^l\left(\boldsymbol{z}^l\right) \\ \boldsymbol{z}^{L+1} &= W^{L+1} \boldsymbol{h}^L + \boldsymbol{b}^{L+1} \\ \boldsymbol{y} = \boldsymbol{h}^{L+1} &= \boldsymbol{f}^{L+1}\left(\boldsymbol{z}^{L+1}\right) \end{aligned} \tag{4-1}$$

The number of total layers (hidden layers and output layer) is referred to as the *depth* of a neural network and the number of neurons per layer is called the *width*.

4-1-1 Activation functions

To introduce nonlinearities in the neural network to increase its expressiveness, activation functions are used. There is a large number of different activation functions each with different advantages. Some widely used activation functions discussed in [30] are

- Linear activation function
$$f(x) = ax$$
- Sigmoid function
$$f(x) = \frac{1}{1 + e^{-x}}$$
- Tanh function
$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$
- Softsign function
$$f(x) = \left(\frac{x}{|x| + 1}\right)$$
- Exponential Linear Unit (ELU) [31]
$$\begin{aligned} f(x) &= x, x >= 0 \\ f(x) &= a\left(e^x - 1\right), x < 0 \end{aligned}$$

- Rectified Linear Unit (ReLU) function

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

  which is a MMPS function
- Gaussian Error Linear Unit (GELU) [32]

$$f(x) = x \cdot \frac{1}{2}[1 + \text{erf}(x/\sqrt{2})]$$

- Scaled Exponential Linear Unit (SELU) [33]

$$f(x) = \lambda x \text{ if } x \geq 0$$
$$f(x) = \lambda\alpha(e^x - 1) \text{ if } x < 0$$

  with $\alpha \approx 1.6733$ and $\lambda \approx 1.0507$.
- Hard sigmoid function [34]

$$f(x) = \max\left(0, \min\left(1, \frac{(x+1)}{2}\right)\right)$$

  This is an approximation of the sigmoid function and is also an MMPS function
- Swish function

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

- SoftMax function

$$f(x)_i = \frac{e^{x_i}}{\sum_{j=1}^{n_l} e^{x_j}}$$
$$\text{for } i = 1, \ldots, n_l$$

  and $\mathbf{x} = (x_1, \ldots, x_{n_l}) \in \mathbb{R}^{n_l}$ and n_l the number of neurons in layer l
- Softplus function [35]

$$f(x) = \log(1 + e^x)$$

  This works as a smooth approximation of the ReLU function.

All these activation functions have different situations where they are more suited. Some activation functions, such as linear activation or SoftMax activation, are often only used in the last layer of a neural network. This makes sure the neural network has the desired behavior. For example, a neural network with only Relu activations can not have a negative output, but if the last layer has a linear activation, it can give negative output.

The ReLU, Hard sigmoid, and Linear activations fit within the MMPS structure given by Definition 2.1. These produce an MMPS function and are therefore equivalent to PWA functions.

4-1-2 Min-Max-Plus neural networks

Min-max-plus neural networks as introduced in [5] are a specific type of neural network, and can consist of three types of layers: linear layers, min-plus layers, or max-plus layers. These specific layers are "linear" in their respective algebra. Using only a combination of max and

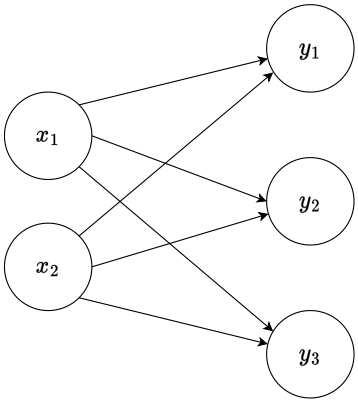

Figure 4-2: Example of a neural network layer I

min layers, the computation time decreases significantly, which makes them interesting to use. An example layer is shown in Figure 4-2, with inputs x_1, x_2 in vector $\boldsymbol{x}$ and outputs y_1, y_2, y_3 in vector $\boldsymbol{y}$. The operations for a layer are defined as:

$$\begin{aligned}
\begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} &= \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \text{ or } \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} \text{ for a linear layer,} \\
\begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} &= \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \end{bmatrix} \otimes \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \text{ for a max-plus layer,} \\
\begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} &= \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \end{bmatrix} \otimes' \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \text{ for a min-plus layer.}
\end{aligned} \tag{4-2}$$

Any layer can be described for inputs $\in \mathbb{R}^n$ and outputs $\in \mathbb{R}^m$ by either a linear transformation $\rho : \mathbb{R}^n \to \mathbb{R}^m$, a max-plus transformation $\alpha : \mathbb{R}^n \to \mathbb{R}^m$, or a min-plus transformation $\beta : \mathbb{R}^n \to \mathbb{R}^m$. These can be described by $\rho(\boldsymbol{x}) = L \cdot \boldsymbol{x}$, $\alpha(\boldsymbol{x}) = A \otimes \boldsymbol{x}$ and $\beta(\boldsymbol{x}) = B \otimes' \boldsymbol{x}$ respectively, with $x \in \mathbb{R}^n$, $L \in \mathbb{R}^{m \times n}$, $A \in \mathbb{R}_{\varepsilon}^{m \times n}$ and $B \in \mathbb{R}_{\top}^{m \times n}$. Here the $\otimes$ operations and extended $\mathbb{R}$ domains are the same as in Section 2-3. The transformations require the output to be in $\mathbb{R}^m$, so there should be at least one entry in each row of $A \neq \varepsilon$ and one entry in each row of $B \neq \top$. This makes sure all transformations are linear in their respective algebra. This does however not mean they are linear in the conventional plus-times algebra. The max-plus layers and min-plus layers have some additions in their operations, but the linear layer does not. Therefore in some configurations of layers, it is useful to add bias to a linear layer, so the output of the neural network can be nonzero for zero inputs.

4-1-3 Neural networks as max-min-plus system

A neural network may contain a section of only max plus and min plus layers. Two of these examples are shown in Figure 4-3. Such a section can be considered as a max-min-plus system, defined the same as a max-min-plus-scaling system as 2.1, but without the scaling operation. A set of only max-plus and min-plus multiplications can be transformed into a conjunctive

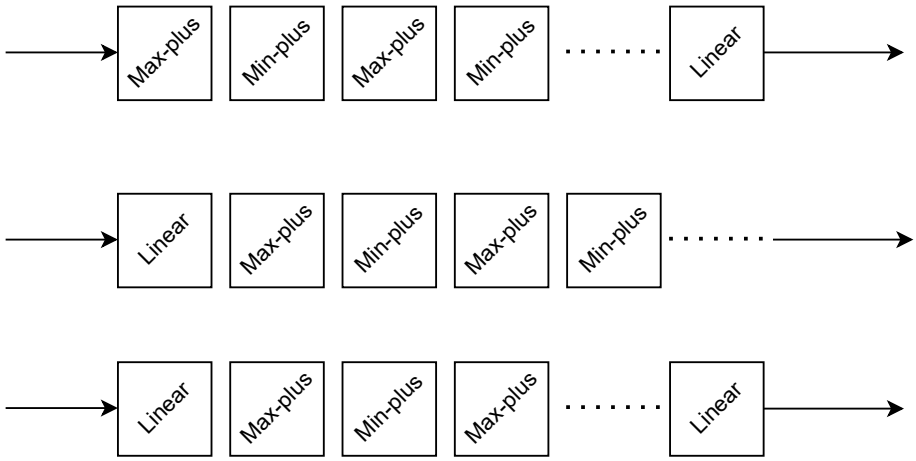

Figure 4-3: Different neural network structures

or disjunctive form as defined in Definition 2.3 and 2.4, which suggests that there is a way to write the system as a maximization of a minimization or vice versa. It is shown in section 4.2 of [36] that a max-min-plus system in general form:

$$z(k+1) = f_{mmp}(z(k)), \quad k = 0, 1, 2, \dots \tag{4-3}$$

can be split into an alternative form, and with $z(k), y(k) \in \mathbb{R}^{n\times n}$ and $C \in \mathbb{R}^{n\times n}$, $A \in \mathbb{R}_{\varepsilon}^{n\times n}$ and $B \in \mathbb{R}_{\top}^{n\times n}$:

$$y(k) = B \otimes' z(k), \quad z(k+1) = A \otimes y(k), \quad k = 0, 1, 2, \dots \tag{4-4}$$

which we can write as

$$z(k+1) = A \otimes \left(B \otimes' z(k)\right), \quad k = 0, 1, 2, \dots \tag{4-5}$$

This result can reduce a section of multiple max-plus and min-plus layers to two operations using only max-plus or min-plus, though it increases the size of the required matrices for these multiplications significantly. If we now apply it to the systems of Figure 4-3 to add scaling, starting with the system depicted at the top we can write the forward computation of the neural network as

$$z(k+1) = C \cdot \left(A \otimes \left(B \otimes' z(k)\right)\right), \quad k = 0, 1, 2, \dots \tag{4-6}$$

Here conventional matrix multiplication is depicted with the $\cdot$ symbol. This system can not represent every PWA function, since applying scaling after a max or min operation can not influence at what point which function is active. If we have

$$\begin{aligned} \alpha \in \mathbb{R}, f_1(x), f_2(x) \in \mathbb{R}_{\varepsilon} \\ \alpha \cdot \max(f_1(x), f_2(x)) \end{aligned} \tag{4-7}$$

then α has no influence on when $f_1(x) > f_2(x)$ if f_1 and f_2 are linear functions of x. The other two networks however can be used to represent any PWA function with enough parameters. They can be described by

$$z(k+1) = A \otimes \left(B \otimes' \left(C \cdot z(k)\right)\right), \quad k = 0, 1, 2, \dots \tag{4-8}$$

for the middle system and if we introduce another matrix $D \in \mathbb{R}^{n \times n}$ we can describe the bottom network with

$$z(k+1) = D \cdot \left(A \otimes \left(B \otimes' \left(C \cdot z(k)\right)\right)\right), \quad k = 0, 1, 2, \ldots \tag{4-9}$$

Often neural networks make use of a different activation in their last layer to get the correct output shape. This is often a linear layer for function approximations, because if you use a max o

4-2 Linear regions

Neural networks can have different degrees of complexity, a very complex neural network with tropical activation functions, such as ReLU, has many different regions on which a different linear function is active. This is called a linear region. Consider the structure given in Equation 4-1, with $\boldsymbol{f}^{1,2,..L}$ as ReLU functions and $\boldsymbol{f}^{L+1}s$ as a linear function $\boldsymbol{f}^{L+1}(x) = x$. The equation becomes

$$\begin{aligned} \boldsymbol{h}^1 &= \max\left(0, W^1\boldsymbol{x} + \boldsymbol{b}^1\right) \\ \boldsymbol{h}^l &= \max\left(0, W^l\boldsymbol{h}^{l-1} + \boldsymbol{b}^l\right) \\ \boldsymbol{y} &= W^{L+1}\boldsymbol{h}^L + \boldsymbol{b}^{L+1}. \end{aligned} \tag{4-10}$$

now for each layer l we define a set $S^l(\boldsymbol{x}) \subseteq \{0, 1, \ldots, n_l\}$ with $i \in S^l$ only if $h_i^l \neq 0$, so only if the function is *active*. Next, collect these into the set $\mathcal{S}(\boldsymbol{x}) = \{S^1(\boldsymbol{x}), S^2(\boldsymbol{x}), \ldots, S^L(\boldsymbol{x})\}$, which is called the activation pattern for a set of inputs $\boldsymbol{x}$. Now use the same definition as [37] for a linear region.

Definition 4.1. *Linear region: for a piecewise linear function $f : \mathbb{R}^{n_0} \to \mathbb{R}^m$ represented by a deep neural network, a linear region is a set of inputs that corresponds to the same activation pattern in the deep neural network.*

The maximum number of regions n hyperplanes divide a d-dimensional space is shown by [38] to be

$$\sum_{s=0}^{d} \binom{n}{s} \tag{4-11}$$

which corresponds to the maximum number of regions of a single layer of a neural network with d as input dimension and s ReLU functions. This can be summed up for every layer to obtain an upper bound. Recently tighter bounds of the number of linear regions for multi-layer networks have been found. [39] obtained an upper bound of 2^N with N the number of total units across layers. This result was improved by [40] assuming the number of neurons is the same in every layer $n_l = n$ and $n_0 = \mathcal{O}(1)$, they found that an upper bound for the number of linear regions is $\mathcal{O}\left(n^{Ln_0}\right)$. This bound was made tighter by [41] to: $\prod_{l=1}^{L} \sum_{j=0}^{d_l} \binom{n_l}{j}$ with $d_l = \min(n_0, n_1, \ldots, n_L)$. Afterwards, this was improved even more by [37], as they achieved the upper bound:

$$\sum_{(j_1,\ldots,j_L)\in J} \prod_{l=1}^{L} \binom{n_l}{j_l}$$

With $J = \{(j_1, \ldots, j_L) \in \mathbb{Z}^L\}$ such that $0 \leq j_l \leq \min(n_0, n_1 - j_1, n_1 - j_2, \ldots n_{l-1} - j_{l-1}, n_L)$ for $l = 1, 2, \ldots, L$.

There are lower bounds too, given by [39]

$$\left(\prod_{l=1}^{L-1} \text{floor}\left(\frac{n_l}{n_0}\right)^{n_0}\right) \sum_{j=0}^{n_0} \binom{n_L}{j} \tag{4-12}$$

when $n_l > n_0$. Another lower bound is given by [37] as:

$$\left(\prod_{l=1}^{L-1} \left(\text{floor}\left(\frac{n_l}{n_0}\right) + 1\right)^{n_0}\right) \sum_{j=0}^{n_0} \binom{n_L}{j} \tag{4-13}$$

with the more restrictive constraint $n_l > 3n_0$.

There are ways to compute the exact number of linear regions. [37] proposes the use of a systematic method to count integer solutions, namely the onetree approach [42]. This method counts the number of solutions to a mixed integer optimization problem defined in [37]. This mixed integer problem can be quite complex to compute, therefore the use of the bounds proposed could prove more efficient. Having an idea of how many linear regions a neural network can describe is useful when you have more information about a PWA control law. If you have an idea of how many linear regions your PWA function has, you have a lower bound on how many linear regions you need to approximate this function.

4-3 Training neural networks

Training a neural network means fitting the weights and biases in such a way that they accurately portray a function that maps from the given training inputs to the training outputs.

To train a neural network, first, a metric for performance: a cost function, sometimes called a loss function is required. Given a set of recorded inputs x and measured outputs $\boldsymbol{y}$. The output of the neural network is an approximation of the measurements, as a function of the weights and biases: $\hat{\boldsymbol{y}} = \boldsymbol{f}\left(W^{1,2,\ldots L}, \boldsymbol{b}^{1,2,\ldots L}, \boldsymbol{x}\right)$. Once the network is trained the estimate is simply $\hat{\boldsymbol{y}} = \boldsymbol{f}(\boldsymbol{x})$. Let a cost function, for example, a mean squared error, be $C(W, \boldsymbol{b}) = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$. The goal of training a neural network is to minimize this cost function over the weights and biases.

$$\underset{W^{1,2,\ldots L}, \boldsymbol{b}^{1,2,\ldots L}}{\text{minimize}} C(\boldsymbol{W}, \boldsymbol{b}) \tag{4-14}$$

In general, this is a nonlinear and non-convex optimization problem. This optimization problem can be solved by using variants of gradient descent.

The process of training with gradient descent-based methods is described in Figure 4-4. First, with an initialized set of weights and biases, the estimate $\hat{y}$ is put into the cost function and a cost is computed. These steps are called the forward pass. Then the gradient of the error with respect to the weights and biases is computed. For deep neural networks, this is often done with a process called back-propagation, which will be discussed further in Section 4-3-1. With this gradient, an update rule is performed and the weights and biases are updated. These steps are called the backward pass. There are different update rules for different gradient descent-based methods:

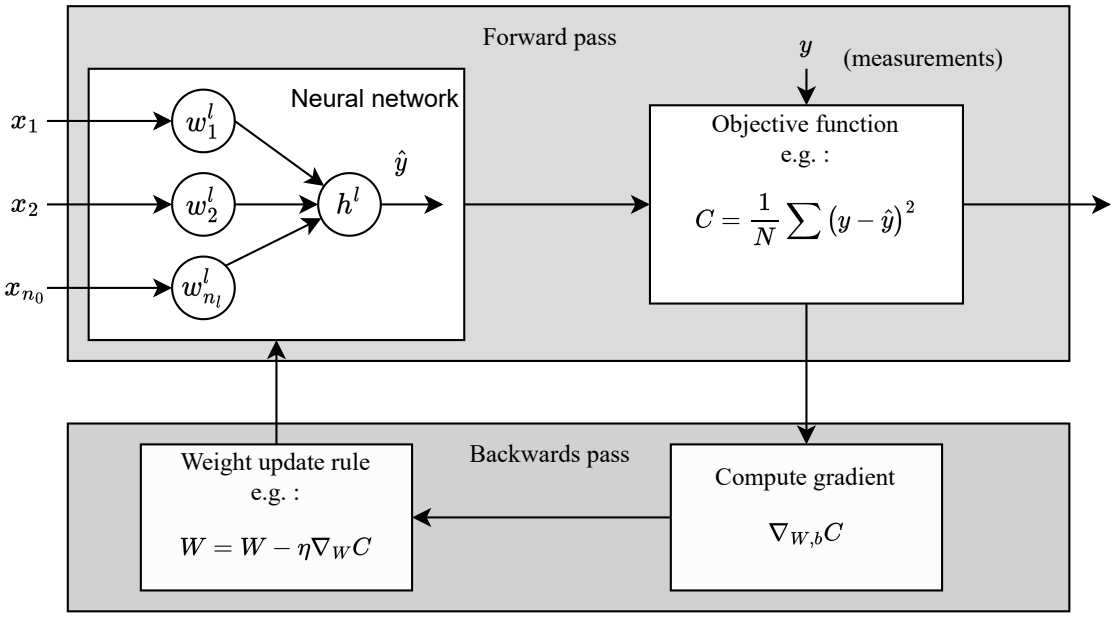

Figure 4-4: Training of a neural network.

4-3-1 Back-propagation

To compute the gradient for a deep neural network, an algorithm called back-propagation [43] is often used. The goal of this algorithm is to find the gradient of a cost function C with respect to the weights and biases:

$$\boldsymbol{\nabla}_{w,b} C = \left(\frac{\partial C}{\partial w_{1,1}^1}, \frac{\partial C}{\partial w_{1,2}^1}, \cdots, \frac{\partial C}{\partial w_{n_{L+1},m}^{L+1}}, \frac{\partial C}{\partial b_1^1}, \frac{\partial C}{\partial b_2^1}, \cdots, \frac{\partial C}{\partial b_{n_{L+1}}^{L+1}} \right)^T \tag{4-15}$$

Instead of writing this as one vector, we can retain the shapes of the matrices by writing it as the tuple:

$$\boldsymbol{\nabla}_{w,b} C = \left(\frac{\partial C}{\partial W^1}, \frac{\partial C}{\partial W^2}, \cdots \frac{\partial C}{\partial W^{L+1}}, \frac{\partial C}{\partial \boldsymbol{b}^1}, \frac{\partial C}{\partial \boldsymbol{b}^2}, \cdots, \frac{\partial C}{\partial \boldsymbol{b}^{L+1}} \right) \tag{4-16}$$

with $\frac{\partial C}{\partial W^l} \in \mathbb{R}^{n_l \times n_{l-1}}$ and $\frac{\partial C}{\partial \boldsymbol{b}^l} \in \mathbb{R}^{n_l \times 1}$. A cost function to optimize for a neural network given a set of measure outputs is $\boldsymbol{y}$

$$C\left(W, \boldsymbol{b}\right) = \frac{1}{N} \sum_{i=1}^{N} \left(\boldsymbol{y}_i - \hat{\boldsymbol{y}}_i\right)^2 \tag{4-17}$$

This is an example of a cost function, there are other cost functions as well, some using different norms such as the ∞-norm or the 1-norm. Now for each layer, consider an auxiliary variable δ^l for $l = 1, 2, ..., L+1$ defined as

$$\boldsymbol{\delta}^l = \frac{\partial C}{\partial \boldsymbol{z}^l}. \tag{4-18}$$

For the last layer, we need the derivative with respect to the output of the neural network. This is given by:

$$\frac{dC}{d\boldsymbol{h}^{L+1}} = \frac{dC}{d\hat{\boldsymbol{y}}} = \hat{\boldsymbol{y}} - \boldsymbol{y} \tag{4-19}$$

Using this derivative, we can compute the δ^{L+1} for the last layer as:

$$\begin{aligned} \boldsymbol{\delta}^{L+1} &= \frac{\partial C}{\partial \boldsymbol{h}^{L+1}} \frac{\partial \boldsymbol{h}^{L+1}}{\partial \boldsymbol{z}^{L+1}} \\ &= \frac{\partial C}{\partial \boldsymbol{h}^{L+1}} \circ \boldsymbol{f}'_{L+1}\left(\boldsymbol{z}^{L+1}\right) \\ &= (\hat{\boldsymbol{y}} - \boldsymbol{y}) \circ \boldsymbol{f}'_{L+1}\left(\boldsymbol{z}^{L+1}\right). \end{aligned} \tag{4-20}$$

Here $\circ$ denotes the element-wise product. For the hidden layers, this becomes

$$\begin{aligned} \boldsymbol{\delta}^{l} &= \frac{\partial C}{\partial \boldsymbol{z}^{l}} \\ &= \frac{\partial C}{\partial \boldsymbol{z}^{l+1}} \frac{\partial \boldsymbol{z}^{l+1}}{\partial \boldsymbol{z}^{l}} \\ &= \frac{\partial \boldsymbol{z}^{l+1}}{\partial \boldsymbol{z}^{l}} \boldsymbol{\delta}^{l+1} \end{aligned} \tag{4-21}$$

recall that $\boldsymbol{z}^{l+1} = W^{l+1}\boldsymbol{h}^{l} + \boldsymbol{b}^{l+1} = W^{l+1}\boldsymbol{f}_{l+1}\left(\boldsymbol{z}^{l}\right) + \boldsymbol{b}^{l+1}$. Now differentiate with respect to $\boldsymbol{z}^{l}$ to obtain

$$\frac{\partial \boldsymbol{z}^{l+1}}{\partial \boldsymbol{z}^{l}} = W^{l+1}\boldsymbol{f}'_{l+1}\left(\boldsymbol{z}^{l}\right) \tag{4-22}$$

With this result, the rest of the $\boldsymbol{\delta}^{l}$ variables can be iteratively computed using the previous result as:

$$\boldsymbol{\delta}^{l} = (W^{l+1})^{T}\boldsymbol{\delta}^{l+1} \circ \boldsymbol{f}'_{l+1}\left(\boldsymbol{z}^{l}\right) \tag{4-23}$$

untill l=1.

With these iteratively computed auxiliary variables, getting the partial derivative of C with respect to all the weights and biases is straightforward.

$$\begin{aligned} \frac{\partial C}{\partial \boldsymbol{b}^{l}} &= \frac{\partial C}{\partial \boldsymbol{z}^{l}} \frac{\partial \boldsymbol{z}^{l}}{\partial \boldsymbol{b}^{l}} \\ &= \boldsymbol{\delta}^{L} \circ 1 \\ &= \boldsymbol{\delta}^{l} \end{aligned} \tag{4-24}$$

and

$$\begin{aligned} \frac{\partial C}{\partial W^{l}} &= \frac{\partial C}{\partial \boldsymbol{z}^{l}} \frac{\partial \boldsymbol{z}^{l}}{\partial W^{l}} \\ &= \boldsymbol{\delta}^{l} \circ \boldsymbol{h}^{l-1} \\ &\left(= \boldsymbol{\delta}^{l} \circ \boldsymbol{x} \text{ for } l = 1\right) \end{aligned} \tag{4-25}$$

Now we can compute the partial derivative of the cost function with respect to all the weights and biases. If we put them back into a vector form, we obtain the gradient of the neural network.

4-3-2 Update rules

Given a cost function $C(\theta)$ and parameter vector θ there are different update rules [44] in the backward pass step to optimize this cost function. They all use the gradient that is obtained in the back-propagation algorithm. One option is to update the gradient with a set learning rate η. For iteration t with $t \in \mathbb{Z}^+$

$$\theta_t = \theta_{t-1} - \eta \nabla_\theta C(\theta)|_{\theta=\theta_{t-1}} \tag{4-26}$$

if this happens for the full data set, all available inputs X and y, it is called batch gradient descent. This method however is rather slow as it recomputes the gradient for every update, and it requires a lot of memory, which can cause problems when the neural network is big enough. It also does not allow for online learning, where new samples come in in training. It converges to a global minimum for convex problems and a local minimum for non-convex problems. Finding a good local minimum is difficult with this method.

Another option is stochastic gradient descend (SGD), where the updates happen separately for every input x_i and corresponding outputs y_i :

$$\theta_{t,i} = \theta_{t-1,i} - \eta \nabla_\theta C(\theta; x_i; y_i)|_{\theta=\theta_{t-1}}. \tag{4-27}$$

This method is much faster than batch gradient descent, as the same gradient can be used for every sample update. This method also allows for online learning. One problem is that this method is not guaranteed to converge to a minimum, as it will sometimes overshoot the minimum. This is sometimes advantageous as it allows the algorithm to escape local valleys to obtain a better local minimum, which works well when the learning rate η decreases over time.

A combination of these two methods is minibatch gradient descent. This uses a smaller shuffled batch of inputs to perform batch gradient descent each iteration.

$$\theta_{t,i} = \theta_{t-1,i} - \eta \nabla_\theta C(\theta; x_{i:i+n}; y_{i:i+n})|_{\theta=\theta_{t-1}}. \tag{4-28}$$

This makes sure there is a more stable convergence to local minima, while still allowing the algorithm to escape some local minima to find better ones. This method also still allows for online learning. There are still some difficulties, even in minibatch gradient descent. Choosing the correct learning rate to not overshoot the optimum is one of them. Often learning rate schedule, that reduces η over time is used to prevent this. The learning rate parameters can be adjusted to reach an optimum within fewer iterations, but if it becomes too large it can overshoot the optimal θ^*. [44] gives an overview of different update rules.

The gradient descent method with a set learning rate or learning rate schedule has some variations. One method uses a *momentum* approach. This is similar to a ball rolling off a hill, building momentum if the hill is steep. If the previous gradient descent step was large, take another larger step in that direction. this accelerates the process. With a $\gamma \in (0, 1]$, typically set to about 0.9. The γ acts as a sort of terminal velocity, to not infinitely increase the step size. The rule becomes:

$$\begin{aligned} v_t &= \gamma v_{t-1} + \eta \nabla_\theta C(\theta_{t-1})|_{\theta=\theta_{t-1}} \\ \theta_t &= \theta_{t-1} - vt \end{aligned} \tag{4-29}$$

A more complex method is the *Nesterov's accelerated method* [45], which uses a similar principle to the momentum approach, but uses the gradient at the position of what the parameters will be after taking the step. This way it is harder to overshoot the optimal θ^*

$$\begin{aligned} v_t &= \gamma v_{t-1} + \eta \nabla_\theta C\left(\theta - \gamma v_{t-1}\right)|_{\theta=\theta_{t-1}} \\ \theta_t &= \theta_{t-1} - v_t \end{aligned} \tag{4-30}$$

Furthermore, there is *Adagrad* [46], which uses an adaptive learning rate for each entry of θ. The idea behind this is that there is a different learning rate for different features of θ, some of which change more frequently and require a smaller learning rate than others. Let the adaptive gradient be defined by:

$$g_{t,i} = \nabla_\theta C\left(\theta_{t,i}\right)|_{\theta_i=\theta_{t-1,i}} \tag{4-31}$$

for each feature i of θ.Now update the parameters by:

$$\theta(t+1) = \theta(t) - \frac{\eta}{\sqrt{G(t)+\epsilon}} \circ g(t) \tag{4-32}$$

where G(t) a diagonal matrix with its diagonal entries i, i set to the sum of the gradients g_i squared up until step t and ϵ a small number (10^{-9}) to avoid numerical errors. A problem with this method is that the learning rate diminishes quickly. For this problem, another method has been developed: Adadelta. Adadelta uses a running average of the gradient, instead of the sum until time t. This running average is computed with:

$$E[g^2]_{(t)} = \gamma E[g^2]_{(t-1)} + (1-\gamma) g(t)^2 \tag{4-33}$$

Where again $\gamma \in (0, 1]$, typically set to about 0.9.

A modern widely used algorithm is Adaptive Moment Estimation (Adam) [47], which combines some of the previous concepts of adaptive gradients, and momentum.

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1-\beta_1)\, g_t \\ v_t &= \beta_2 v_{t-1} + (1-\beta_2)\, g_t^2 \end{aligned} \tag{4-34}$$

m_t estimates the first moment (mean) of the gradient and v_t the second moment (the uncentered variance). When these variables are initialized at zero, or when the decay rates are small (β_1, β_2 close to 1), or in the earlier steps of the optimization, these variables are biased towards zero. To counteract this they are bias-corrected first with

$$\begin{aligned} \hat{m}_t &= \frac{m_t}{1-\beta_1^t} \\ \hat{v}_t &= \frac{v_t}{1-\beta_2^t} \end{aligned} \tag{4-35}$$

Now similar to Adagrad and Adadelta, the update rule is

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t}+\epsilon}\hat{m}_t \tag{4-36}$$

which works well in practice.

4-4 Exact representation

Multilayer neural networks are known from [48] to be universal function approximators. With no bounds on the depth and width of a neural network, any continuous function can be arbitrarily well approximated, and with infinite depth and width, a function can be perfectly represented. However, we want to perfectly represent it with some tighter bounds on the depth of the network. From [15] we know that the MMPS and PWA functions are equivalent and [20] shows that MMPS functions can represent bounded PWA functions with a finite number of terms. For approximating convex piecewise affine functions, Theorem 2 from [49] provides bounds on the required size of the network. Now use the following lemma:

Lemma 4.2. *every scalar continuous PWA function $f(x) : \mathbb{R}^{n_x} \to \mathbb{R}$ can be written as the difference of two convex PWA functions:*

$$f(x) = \gamma(x) - \eta(x)$$

with $\gamma(x) : \mathbb{R}^{n_x} \to \mathbb{R}$ with n_γ linear regions and $\eta(x) : \mathbb{R}^{n_x} \to \mathbb{R}$ with n_η linear regions.

This is proven by [19], where any scalar PWA function can be written as the difference of convex PWA functions. These convex functions can not only be perfectly represented by neural networks but [49] also gives a minimum requirement for the depth of the neural network. The requirement follows from this lemma:

Lemma 4.3. *Any convex continuous PWA function $g : [0,1]^{n_x} \to \mathbb{R}^+$ that is defined as the pointwise maximum of n_g affine functions*

$$g(x) = \max_{1,2,\dots,n_g} g_i(x),$$

can be exactly represented by a deep ReLU network with $M = n_x + 1$ neurons per layer and a depth n_g.

The proof is given by Theorem 2 of [49]. Furthermore, [50] proves that any piecewise affine control law $\kappa(x) : [0,1]^{n_x} \to \mathbb{R}^{+^{n_u}}$ can be represented by a neural network with a predetermined size. First, split the control law into a PWA function per output dimension and apply Lemma 4.2 such that:

$$\begin{aligned} \kappa_i(x) &: [0,1]^{n_x} \to \mathbb{R}^+ \\ \kappa_i(x) &= \gamma_i(x) - \eta_i(x) \end{aligned} \tag{4-37}$$

with each $\gamma_i(x)$ and $\eta_i(x)$ convex PWA functions. Then by Lemma 4.3, each of these convex PWA functions can be represented by a neural network with parameters $\theta_{\gamma,i}$ or $\theta_{\eta,i}$, depth $r_{\gamma,i}$ or $r_{\eta,i}$ and width $M = n_x + 1$. This results in a vector of neural networks

$$\kappa(x) = \begin{bmatrix} \mathcal{N}(x; \theta_{\gamma,1}, M, r_{\gamma,1}) - \mathcal{N}(x; \theta_{\eta,1}, M, r_{\eta,1}) \\ \vdots \\ \mathcal{N}(x; \theta_{\gamma,n_u}, M, r_{\gamma,n_u}) - \mathcal{N}(x; \theta_{\eta,n_u}, M, r_{\eta,n_u}) \end{bmatrix}. \tag{4-38}$$

This works for explicit control laws that map from $[0,1]^{n_x}$ to $\mathbb{R}^+$, in a more general case we want this to hold for any PWA control law. This is the case for the following assumption:

Assumption 4.4. *There exist two invertable affine transformations* $\mathcal{A}_x : \mathbb{X} \to [0,1]^{n_x}$ *and* $\mathcal{A}_u : \mathbb{U} \to \mathbb{R}^{+^{n_u}}$ *for an explicit control law* $\kappa : \mathbb{X} \to \mathbb{U}$ *such that*

$$\kappa(x) = \mathcal{A}_u^{-1} \circ \tilde{\kappa}(\hat{x})$$

with $\hat{x} = \mathcal{A}_x \circ x$. *These transformations always exist when* $\mathbb{X}$ *and* $\mathbb{U}$ *are compact sets.*

With these transforms and the representation of the altered control law $\tilde{\kappa}(\hat{x}) : [0,1]^{n_x} \to \mathbb{R}^{+^{n_u}}$ existing of $2n_u$ deep neural networks, any PWA control law $\kappa(x)$ can be approximated. A PWA control law arises from a constrained LQR problem, where $\mathbb{X}$ and $\mathbb{U}$ are closed sets, so Assumption 4.4 holds for these problems. With this, any continuous PWA function can be exactly represented with neural networks. If the control law is discontinuous, this does not hold. This exact representation can become quite large, and difficult to obtain due to the nonconvex optimization involved in fitting the neural network to the PWA function. Therefore an approximation could be easier to obtain and faster to use.

Chapter 5

Approximation procedure

The main interest of this research is the approximation of MPC control laws. Approximations are mostly relevant when an explicit MPC control law is not available, but a fast computation is still required. Therefore we will not be using explicit MPC, whereas others sample an explicit control law to obtain data for approximations [6]–[8], we will sample an implicit MPC controller. Other works have seen the use in approximating either nonlinear MPC [10], [51] or linear MPC with a long prediction horizon [52]. We will also consider only simple feedforward neural networks, whereas others consider other structures such as Long short-term memory networks [7], [9] or recurrent neural networks [10]. What is not thoroughly researched yet and we will aim to contribute is an answer to the following questions

- Is there a significant difference in approximating PWA MPC control laws compared to non-PWA MPC control laws?
- Does the max-min-plus-scaling structure of a neural network offer a benefit compared to other neural networks with other activation functions when approximating MPC control laws?
- Do max-plus and min-plus layers in a neural network offer better performance in approximating PWA MPC control laws?
- What is an appropriate sampling strategy to obtain a satisfactory approximation of an MPC control law?

5-1 Making controllers and obtaining control law

In order to answer the main question of this research "What are the various factors that need to be taken into consideration to successfully approximate a PWA MPC control law using neural networks?" We first need to have MPC control laws. For this, we will consider the control of two systems. First of all we consider the nonlinear system of an inverted

pendulum. This system is chosen because it can be described with only two states. This low dimensionality makes it easy to gather enough data to approximate the control law, and makes it easy to visualize the approximate control laws of different neural networks. First, we will linearize the nonlinear system at the equilibrium and design a linear MPC controller. This also gives a PWA control law. Besides this linearized description, we will also use a PWA description. A piecewise affine MPC controller is useful because it is a more accurate representation of the nonlinear system. The downside is, however, that computations for an MPC controller are more complex. The derivative function of this system contains one sine function, which can be approximated with a PWA function. This makes it easy to obtain a PWA description of the system. The resulting control law will also be two-dimensional, which makes it easier to study visually. We will use this PWA description to transform the PWA system into an MLD system in order to construct an MPC controller.

Next, we will consider a system with a double inverted pendulum. This system description is also much more complex, where the single pendulum had only a sine function in one dimension of the derivative function, and a linear description in the rest, this is not the case for the double pendulum. This system has many more sine and cosine functions and other nonlinear behaviors that are difficult to describe with a piecewise affine system description. Applying MPC to such a complicated system with higher dimensionality would also significantly increase the number of PWA regions, making the computation more complex. That is why for this system we will only consider a linearized MPC controller at the upright equilibrium. The nonlinearity of the system also makes it more difficult to control, and any errors in approximating an MPC control law would be more significant. The increased dimensionality also plays a role in getting enough data to sample the system, as it requires sampling in higher dimensional state space.

5-2 Approximating with different parameters

5-2-1 Sampling Strategy

The controllers will be approximated using neural networks with varying parameters. In this study, we assume that an explicit model predictive control (MPC) law is not available, necessitating the development of an implicit strategy to approximate an MPC control law through sampling. Sampling an infinite state space is impossible, so we first select a region of interest. This will be a region based on the physical qualities of the system in which we expect the controller to operate, and some extra margin. We can choose this to be a polyhedral set centered at the origin. Initially, a uniform grid sampling strategy will be employed to generate a sufficiently large dataset, which will then be divided into subsets. We will compare random subsets and sparse subsets of the original dataset to determine if training the network on these subsets yields significant differences. Besides this uniform grid sampling strategy, we will compare a random sampling strategy with a uniform distribution function and a stratified sampling strategy. The stratified sampling strategy will have a higher probability of selecting more points if it finds that nearby points have a larger contrast. This contrast-based region technique is similar to the strategy used in [53] and the goal is to avoid overrepresentation of 'simple' areas, while still having enough representative samples from 'complicated' areas.

These techniques will be compared using a single pendulum system. The two-state nature of this system allows for better visualization and easier determination of stability regions.

For the double pendulum system, a different sampling strategy will be employed. We will simulate trajectories from various initial states to capture the system's dynamics and behavior. In these initial states, we will only vary the position of the two rods, and not the initial angular velocity. Because sampling for a four-dimensional system requires a lot of samples and for some states finding a physical interpretation to find out what states need to be sampled can be complicated. For example, we have a pendulum with two links and want to make sure it stays upright and not moving. Finding out the maximum angle the two rods can be to be stabilized by the controller is still doable, but finding out what initial angular velocities are permissible makes the problem much more complex.

5-2-2 Neural Network training

The main motivation of this research is to find if there is merit to using MMPS activation functions over smooth activation functions when approximating PWA control laws. Alternatively, to see if there is an advantage to approximating MPC control laws in general with MMPS activation functions. The activation functions discussed in Section 4-1-1 are used for many different purposes, but our main focus is on the difference between MMPS activation functions and non-MMPS functions. For this, we will train models with different activation functions in the hidden layers. The output layer will always be a linear activation to make sure that every neural network we train has the ability to output all values in the input space and not only positive values, for example.

The networks will all be trained on the same three datasets. We will keep the number of layers and number of neurons per layer the same. Since we want to evaluate the eventual neural network, and not the performance during training, we need to make sure the neural networks are sufficiently trained. Some activation functions can train with fewer iterations, so to ensure they are all sufficiently trained, we will not have the same exact number of epochs for each network. We will have a maximum of 300 epochs, but we will implement an early stopping approach [54]. After each epoch, we will validate the loss on a separate validation set, and if this validation loss keeps increasing, with a set patience of 10 epochs, we will stop the training and return to weights that caused the lowest loss. This ensures that all neural networks are trained sufficiently, but not overtrained.

We will be using the Adam optimizer, as discussed in Section 4-3-2, on all networks, since this algorithm is computationally efficient and well-suited for this problem. We will also be using the same batch size of 50 on each network.

5-2-3 Implementing max-plus and min-plus layers

Another type of neural network that has not yet been used in approximating MMPS control laws is a neural network with max-plus or min-plus layers. Since this is a more novel neural network, we need to do more of the implementation ourselves, as opposed to using more common neural network libraries. That is why one option is to build a neural network from scratch, using max-plus and min-plus multiplication in its layers. This allows great customization but does have a lot to be implemented. Some steps to be implemented are

- Define initial random weights and biases
- Do a forward pass to compute the current loss with the current weights and biases.
- Compute the gradient with respect to all the weights and biases, either numerically or analytically.
- Use an update rule such as Adam to update the weights and biases.

Through some trials, finding the analytical gradient of a neural network proved to be more difficult than finding a numerical gradient since there are more libraries available for this. Because training requires multiple iterations, it is essential that the steps that are performed in each iteration are performed efficiently. Finding the gradient is the most difficult problem, but this can be done with the JAX autogradient library [55]. This allows the use of a taped gradient, by running multiple iterations, the autogradient watches what computations are done, and sees if there are any optimizations to be found. This adds more overhead but is faster with more iterations. Eventually, testing all these combined steps, the neural network seems to converge to a good fit but ends up in a more noisy state. Changing the step size, batch size, and other network parameters had the same result. This made this network difficult to compare to other results from other standard libraries.

This is why another option is to use the TensorFlow library [56]. Tensorflow is a widely used neural network library that also allows for customization. It allows us to do so by using custom layer types. However, this does require some knowledge of the library about how it handles batch training and predictions, custom data types such as tensors, and its layer structure. In order to handle a custom layer efficiently, we need to do a max-plus multiplication using built-in functions. By using these built-in functions we can reshape and repeat the weights to do an addition and use the reduction function to ensure the right maximization is done and the result has the required shape. With this achieved min-plus multiplication is trivial by using minimization instead of maximization. The code for this is in Appendix B, and now there is a way to allow for max-plus and min-plus layers in our own defined neural networks.

5-3 Comparing results

The main way to assess the performance of an approximation of a function is by checking the error between the function and the approximation. In this case, we will be looking at the root mean squared error. In training a neural network, this happens many times, however only to the training data. We also want to see how much error there is when providing test data that the network has not trained on. We will be using this validation loss as a metric to compare the performance of neural networks. Additionally, we want to see how these differences between the exact control law and approximate control law affect the trajectory. We will do this by simulating trajectories from many different initial states on the nonlinear system. We will compute the cost of a trajectory by using the same cost as the MPC controller that the controllers are approximating. This comes out the same cost as Equation 3-39 with the p-norm the same as the respective controller. This is similar to the method used in [50], [52]. Since the MPC controllers optimize a linearized, or PWA approximation of the system, their control laws are not necessarily optimal for the nonlinear system. Since we want to

compare with the MPC controllers as a benchmark we will normalize the cost of the neural network simulations by the cost of the MPC controller trajectories and subtract 1 to get a single scalar score.

Chapter 6

Case studies

In order to approximate MPC control laws, we first need MPC controllers for systems. This chapter discusses the derivation of the equations of motion for two systems. We will find a linearized approximation of a single inverted pendulum, as well as a PWA approximation of this pendulum. The other system, a double inverted pendulum, will only be using a linearized approximation.

6-1 Linear inverted pendulum

Consider the model of an inverted pendulum:

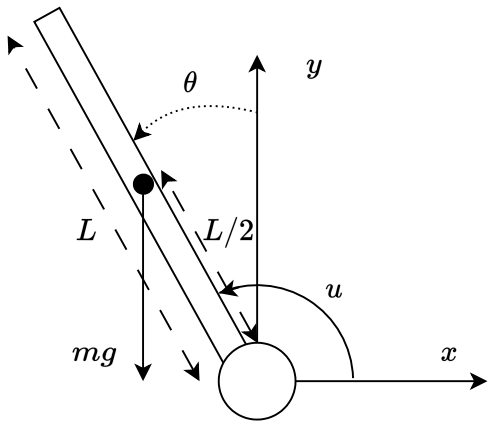

Figure 6-1: Model of an inverted pendulum

All the parameters of this system can be found in Appendix A-2. From this model we can

derive the following equations of motion:

$$\begin{aligned}\dot{\boldsymbol{x}}(t) &= \boldsymbol{f}(\boldsymbol{x}(t), \boldsymbol{u}(t)) \\ \underbrace{\begin{bmatrix}\dot{\theta}(t) \\ \ddot{\theta}(t)\end{bmatrix}}_{\boldsymbol{x}(k)} &= \underbrace{\begin{bmatrix}\dot{\theta}(t) \\ \frac{3g}{2L}\sin\left(\theta(t)\right) + \frac{3}{mL^2}u\end{bmatrix}}_{\boldsymbol{f}(\boldsymbol{x}(t),\boldsymbol{u}(t))}.\end{aligned} \tag{6-1}$$

To linearize this nonlinear system about the upside equilibrium ($\bar{\boldsymbol{x}} = \mathbf{0}, \bar{u} = 0$), we use Jacobian linearization. First we define the matrices $A := \frac{\partial f}{\partial x}\Big|_{\substack{x=\mathbf{0}\\u=0}}$ and $B := \frac{\partial f}{\partial u}\Big|_{\substack{x=\mathbf{0}\\u=0}}$. The continuous time matrices come out to be

$$\begin{aligned}A_c &= \begin{bmatrix}0 & 1 \\ \frac{3g}{2L}\cos\left(\theta(t)\right) & 0\end{bmatrix}\Bigg|_{\dot{\theta}=0,\theta=0} = \begin{bmatrix}0 & 1 \\ \frac{3g}{2L} & 0\end{bmatrix} \\ B_c &= \begin{bmatrix}0 \\ \frac{3}{mL^2}\end{bmatrix}\end{aligned}. \tag{6-2}$$

Next, we need to discretize the matrices. For this, we use a zero-order hold mechanism with a constant timestep h. We can transform the continuous time state matrices A_c and B_c to discrete-time (A_d and B_d) with the following equations [57]:

$$\begin{aligned}A_d &= e^{A_c h} \\ B_d &= A_c^{-1}\left(e^{A_c h} - I\right) B_c.\end{aligned} \tag{6-3}$$

So for constant time step h, the discrete system becomes:

$$\begin{aligned}x(k+1) &= e^{A_c h}x(k) + A_c^{-1}\left(e^{A_c h} - I\right) B_c u(k) = \\ x(k+1) &= A_d x(k) + B_d u(k).\end{aligned} \tag{6-4}$$

Now with these discrete matrices, we can use the method in Section 3-2 to formulate and solve an MPC problem.

This does not yet take into account any constraints of the inputs on the inputs or states. Say we have a limit on the amount of torque we can provide, so for example $|u| \leq 10$, and the states can not exceed $|x| \leq \begin{bmatrix}\pi \\ 10\pi\end{bmatrix}$. We also want our pendulum to end up in its upright position (or close to that position), or at least in a control invariant set $\mathcal{X}_f$. We find this invariant set by finding what states can still enter the set with the maximum control input of 10 or -10. We also place a cost on the states and inputs by defining $Q = \begin{bmatrix}100 & 0 \\ 0 & 1\end{bmatrix}$ and $R = \begin{bmatrix}1\end{bmatrix}$ So we add that to our constraints. The optimization problem becomes:

$$\begin{aligned}
\underset{u(1:N-1)}{\text{minimize}} \quad & V\left(x(0), u(0:N-1)\right) \\
\text{s.t.} \quad & x(k+1) = A_d x(k) + B_d u(k) \quad \forall k = 0, 1, \ldots, N-1 \\
& x(k) \leq \begin{bmatrix} \pi \\ 10\pi \end{bmatrix} \quad \forall k = 0, 1, \ldots, N \\
& -x(k) \leq \begin{bmatrix} \pi \\ 10\pi \end{bmatrix} \quad \forall k = 0, 1, \ldots, N \\
& u(k) \leq 10 \quad \forall k = 0, 1, \ldots, N-1 \\
& -u(k) \leq 10 \quad \forall k = 0, 1, \ldots, N-1 \\
& x(N) \in X_f
\end{aligned} \tag{6-5}$$

, which we can now solve.

6-2 Piecewise affine inverted pendulum

Now consider the same inverted pendulum model as Section 6-1, which results in the same nonlinear equations of motion:

$$\begin{bmatrix} \dot{\theta}(t) \\ \ddot{\theta}(t) \end{bmatrix} = \begin{bmatrix} \dot{\theta}(t) \\ \frac{3g}{2L} \sin\left(\theta(t)\right) + \frac{3}{mL^2} u \end{bmatrix}, \tag{6-6}$$

but instead of using Jacobian linearization, we substitute the sine function with the following:

$$\sin\left(\theta(t)\right) = \alpha_i \theta(t) + \beta_i \quad \forall \theta(t) \in \Omega_i. \tag{6-7}$$

Here each Ω_i is a different affine region. We can use for example 7 regions of equal size on the domain $[-\pi, \pi]$. This would look like Figure 6-2.

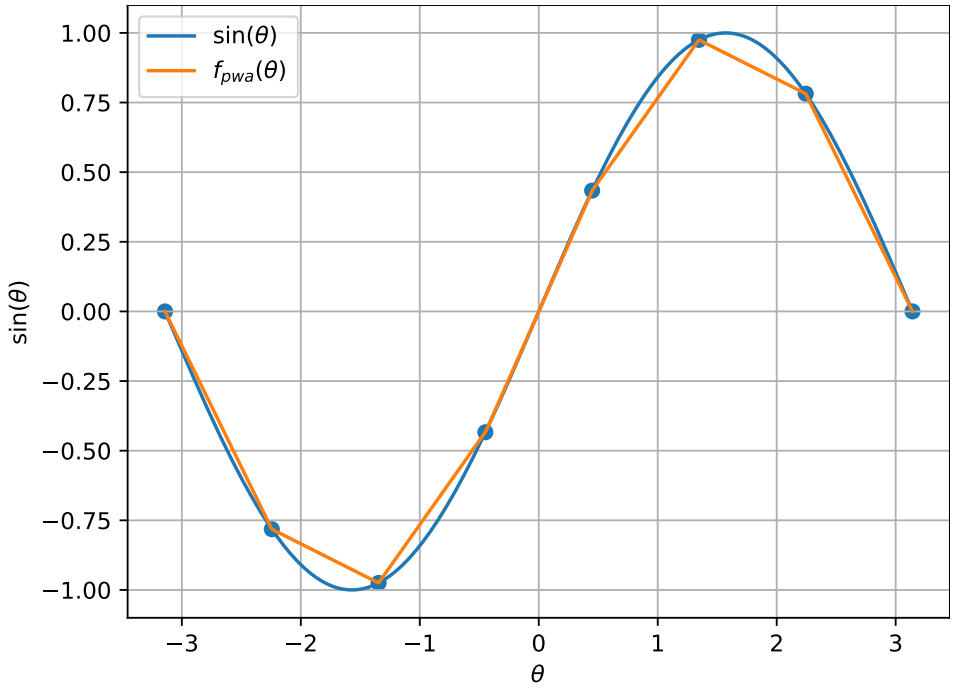

Figure 6-2: Piecewise affine approximation of the sine function with seven affine regions.

The resulting continuous time piecewise affine system is then

$$\underbrace{\begin{bmatrix} \dot{(}t)\theta \\ \ddot{\theta}(t) \end{bmatrix}}_{\dot{x}(t)} = \underbrace{\begin{bmatrix} 0 & 1 \\ \frac{3g}{2L}\alpha_i & 0 \end{bmatrix}}_{A} \begin{bmatrix} \theta(t) \\ \dot{\theta}(t) \end{bmatrix} + \underbrace{\begin{bmatrix} 0 \\ \beta_i \end{bmatrix}}_{B_1} + \underbrace{\begin{bmatrix} 0 \\ \frac{3}{mL^2} \end{bmatrix}}_{B_2} \begin{bmatrix} u(t) \end{bmatrix} \quad \forall \theta \in \Omega_i. \tag{6-8}$$

Now to discretize this with a zero-order hold mechanism, with the extra constant terms B_1 in Equation 6-8 there are also extra terms in the discrete result. The derivation of this discretization method is in Appendix A-1

$$x(k+1) = \underbrace{e^{Ah}}_{A_d} x(k) + \underbrace{A^{-1}\left(e^{Ah} - I\right) B_1}_{B_l d} + \underbrace{A^{-1}\left(e^{Ah} - I\right) B_2}_{B_d} u(k). \tag{6-9}$$

This gives us a piecewise affine system with s regions in the following form:

$$x[k+1] = \begin{cases} A_{d1}x(k) + B_{d1}u(k) + B_{1d1} \text{ if } \theta \in \Omega_1 \\ A_{d2}x(k) + B_{d2}u(k) + B_{1d2} \text{ if } \theta \in \Omega_2 \\ \vdots \\ A_{ds}x(k) + B_{ds}u(k) + B_{1ds} \text{ if } \theta \in \Omega_s. \end{cases} \tag{6-10}$$

With this piecewise affine description of the system, we can use the method described in Section 3-4 to set up an MPC controller.

6-3 Double pendulum

Consider the following double pendulum system: A double pendulum with two rods. Each rod has a mass at the tip, shifting its center of gravity from the center. The position of the

center of mass is denoted by c_i. The pendulum is controlled by a torque u given by a motor to the first rod.

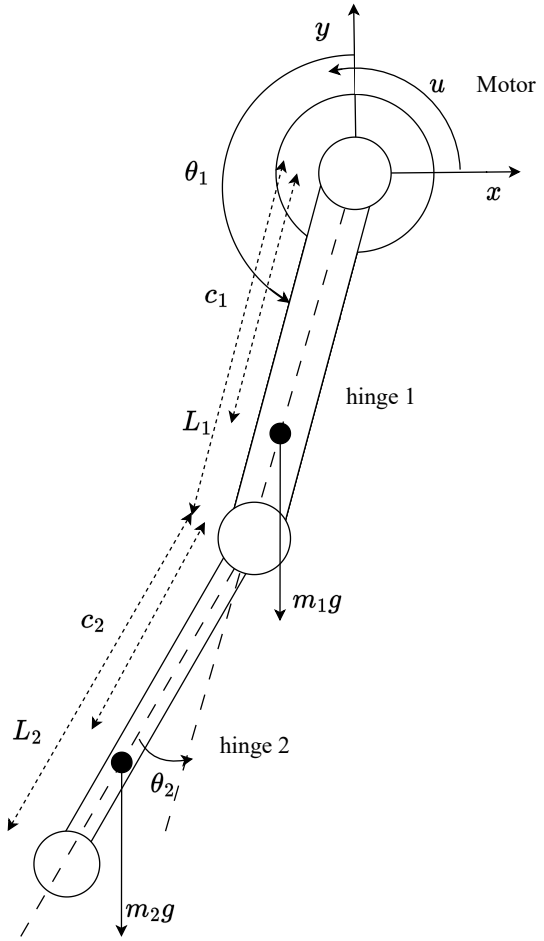

Figure 6-3: Model of a double pendulum

All the parameters of this model are described in Appendix A-2 To obtain the equations of motion in generalized coordinates $\begin{bmatrix}\theta_1 & \theta_2\end{bmatrix}^T$ we use the TMT method [58]. For this we have the equation from [58]

$$\mathbf{T}^T\mathbf{M}\mathbf{T}\ddot{\mathbf{q}} = \mathbf{Q} + \mathbf{T}^T(\mathbf{F} - \mathbf{M}\mathbf{g}) \tag{6-11}$$

and from this, we obtain the equations of motion:

$$\ddot{\mathbf{q}} = \left(\mathbf{T}^T\mathbf{M}\mathbf{T}\right)^{-1}\left(\mathbf{Q} + \mathbf{T}^T(\mathbf{F} - \mathbf{M}\mathbf{g})\right) \tag{6-12}$$

Consider the coordinates of the center of masses of hinge 1 and hinge 2. We can describe the orientation and position of the two bodies with the position of their center of mass (x_1, y_1) and (x_2, y_2), and their rotation around that point ϕ_1 and ϕ_2. But the goal is to describe the system with only two generalized coordinates θ_1 and θ_2. If we write the original

coordinates as a function of these generalized coordinates, we get:

$$T_i = \begin{bmatrix} x_1 \\ y_1 \\ \phi_1 \\ x_2 \\ y_2 \\ \phi_2 \end{bmatrix} = \begin{bmatrix} -c_1 \sin(\theta_1) \\ c_1 \cos(\theta_1) \\ \theta_1 \\ -L_1 \sin(\theta_1) - c_2 \sin(\theta_1 + \theta_2) \\ L_1 \cos(\theta_1) + c_2 \cos(\theta_1 + \theta_2) \\ \theta_1 + \theta_2 \end{bmatrix} \tag{6-13}$$

By taking the jacobian of T_i, we obtain $\mathbf{T}$:

$$\begin{bmatrix} -c_1 \cos(\theta_1) & 0 \\ -c_1 \sin(\theta_1) & 0 \\ 1 & 0 \\ -L_1 \cos(\theta_1) - c_2 \cos(\theta_1 + \theta_2) & -c_2 \cos(\theta_1 + \theta_2) \\ -L_1 \sin(\theta_1) - c_2 \sin(\theta_1 + \theta_2) & -c_2 \sin(\theta_1 + \theta_2) \\ 1 & 1 \end{bmatrix} \tag{6-14}$$

We can set up the mass matrix $\mathbf{M}$

$$\mathbf{M} = \begin{bmatrix} m_1 & 0 & 0 & 0 & 0 & 0 \\ 0 & m_1 & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{L_1^2 m_1}{12} & 0 & 0 & 0 \\ 0 & 0 & 0 & m_2 & 0 & 0 \\ 0 & 0 & 0 & 0 & m_2 & 0 \\ 0 & 0 & 0 & 0 & 0 & \frac{L_2^2 m_2}{12} \end{bmatrix} \tag{6-15}$$

We can find the $\mathbf{g}$ vector by taking the second derivative of the T_i vector:

$$\begin{bmatrix} c_1 \sin(\theta_1) \left(\dot{\theta}_1\right)^2 - c_1 \cos(\theta_1) \ddot{\theta}_1 \\ -c_1 \sin(\theta_1) \ddot{\theta}_1 - c_1 \cos(\theta_1) \left(\dot{\theta}_1\right)^2 \\ \ddot{\theta}_1 \\ c_2 \left(\dot{\theta}_1 + \dot{\theta}_2\right)^2 \sin(\theta_1 + \theta_2) - c_2 \left(\ddot{\theta}_1 + \dot{\theta}_2\right) \cos(\theta_1 + \theta_2) + L_1 \sin(\theta_1) \left(\dot{\theta}_1\right)^2 - L_1 \cos(\theta_1) \ddot{\theta}_1 \\ -c_2 \left(\dot{\theta}_1 + \dot{\theta}_2\right)^2 \cos(\theta_1 + \theta_2) - c_2 \left(\ddot{\theta}_1 + \dot{\theta}_2\right) \sin(\theta_1 + \theta_2) - L_1 \sin(\theta_1) \ddot{\theta}_1 - L_1 \cos(\theta_1) \left(\dot{\theta}_1\right)^2 \\ \ddot{\theta}_1 + \dot{\theta}_2 \end{bmatrix} \tag{6-16}$$

and setting the accelerations of the generalized coordinates to zero:

$$\mathbf{g} = \begin{bmatrix} c_1 \sin(\theta_1) \left(\dot{\theta}_1\right)^2 \\ -c_1 \cos(\theta_1) \left(\dot{\theta}_1\right)^2 \\ 0 \\ c_2 \left(\dot{\theta}_1 + \dot{\theta}_2\right)^2 \sin(\theta_1 + \theta_2) + l_1 \sin(\theta_1) \left(\dot{\theta}_1\right)^2 \\ -c_2 \left(\dot{\theta}_1 + \dot{\theta}_2\right)^2 \cos(\theta_1 + \theta_2) - l_1 \cos(\theta_1) \left(\dot{\theta}_1\right)^2 \\ 0 \end{bmatrix} \tag{6-17}$$

The forces acting on the system can be described by the vector **F**:

$$\mathbf{F} = \begin{bmatrix} 0 \\ -gm_1 \\ 0 \\ 0 \\ -gm_2 \\ 0 \end{bmatrix} \tag{6-18}$$

And the control forces described in the generalized coordinates are:

$$\mathbf{Q} = \begin{bmatrix} u \\ 0 \end{bmatrix} \tag{6-19}$$

Putting all these matrices in Equation 6-12 yields:

$$\begin{aligned}\ddot{\theta}_1 = f_{\ddot{\theta}_1}(q) = \\ \frac{c_2 m_2 \left(-L_1 \sin(\theta_2)\left(\dot{\theta}_1\right)^2 + g\sin(\theta_1+\theta_2)\right)\left(-144L_1c_2\cos(\theta_2) - 12L_2^2 - 144c_2^2\right)}{L_1^2L_2^2m_1 + 12L_1^2L_2^2m_2 + 12L_1^2c_2^2m_1 - 144L_1^2c_2^2m_2\cos^2(\theta_2) + 144L_1^2c_2^2m_2 + 12L_2^2c_1^2m_1 + 144c_1^2c_2^2m_1} + \\ \frac{\left(12L_2^2 + 144c_2^2\right)\left(2L_1c_2m_2\sin(\theta_2)\dot{\theta}_1\dot{\theta}_2 + L_1c_2m_2\sin(\theta_2)\left(\dot{\theta}_2\right)^2 + L_1gm_2\sin(\theta_1) + c_1gm_1\sin(\theta_1) + c_2gm_2\sin(\theta_1+\theta_2) + u\right)}{L_1^2L_2^2m_1 + 12L_1^2L_2^2m_2 + 12L_1^2c_2^2m_1 - 144L_1^2c_2^2m_2\cos^2(\theta_2) + 144L_1^2c_2^2m_2 + 12L_2^2c_1^2m_1 + 144c_1^2c_2^2m_1}\end{aligned} \tag{6-20}$$

and

$$\begin{aligned}\ddot{\theta}_2 = f_{\ddot{\theta}_2}(q) = \\ \frac{c_2m_2\left(-L_1\sin(\theta_2)\left(\dot{\theta}_1\right)^2 + g\sin(\theta_1+\theta_2)\right)\left(12L_1^2m_1 + 144L_1^2m_2 + 288L_1c_2m_2\cos(\theta_2) + 12L_2^2m_2 + 144c_1^2m_1 + 144c_2^2m_2\right)}{L_1^2L_2^2m_1m_2 + 12L_1^2L_2^2m_2^2 + 12L_1^2c_2^2m_1m_2 - 144L_1^2c_2^2m_2^2\cos^2(\theta_2) + 144L_1^2c_2^2m_2^2 + 12L_2^2c_1^2m_1m_2 + 144c_1^2c_2^2m_1m_2} + \\ \frac{\left(-144L_1c_2\cos(\theta_2) - 12L_2^2 - 144c_2^2\right)\left(2L_1c_2m_2\sin(\theta_2)\dot{\theta}_1\dot{\theta}_2 + L_1c_2m_2\sin(\theta_2)\left(\dot{\theta}_2\right)^2 + L_1gm_2\sin(\theta_1) + \right)}{L_1^2L_2^2m_1 + 12L_1^2L_2^2m_2 + 12L_1^2c_2^2m_1 - 144L_1^2c_2^2m_2\cos^2(\theta_2) + 144L_1^2c_2^2m_2 + 12L_2^2c_1^2m_1 + 144c_1^2c_2^2m_1} + \\ \frac{\left(-144L_1c_2\cos(\theta_2) - 12L_2^2 - 144c_2^2\right)\left(c_1gm_1\sin(\theta_1) + c_2gm_2\sin(\theta_1+\theta_2) + u\right)}{L_1^2L_2^2m_1 + 12L_1^2L_2^2m_2 + 12L_1^2c_2^2m_1 - 144L_1^2c_2^2m_2\cos^2(\theta_2) + 144L_1^2c_2^2m_2 + 12L_2^2c_1^2m_1 + 144c_1^2c_2^2m_1}\end{aligned} \tag{6-21}$$

We now have a nonlinear model of the system.

6-3-1 Linear Model

From the nonlinear model of the system, we can get a linear model using Jacobian linearization. First we define the matrices $A := \left.\frac{\partial f}{\partial x}\right|_{\substack{x=\mathbf{0}\\u=0}}$ and $B := \left.\frac{\partial f}{\partial u}\right|_{\substack{x=\mathbf{0}\\u=0}}$ we have our function f:

$$f(x) = \begin{bmatrix} \dot{\theta}_1 \\ \dot{\theta}_2 \\ f_{\ddot{\theta}_1}(x) \\ f_{\ddot{\theta}_2}(x) \end{bmatrix} \tag{6-22}$$

by taking the jacobian we obtain:

$$A := \left.\frac{\partial f}{\partial x}\right|_{\substack{x=\mathbf{0}\\u=0}} = \begin{bmatrix} \vdots & \vdots & 1 & 0 \\ \mathbf{v_1} & \mathbf{v_2} & 0 & 1 \\ \vdots & \vdots & 0 & 0 \\ \vdots & \vdots & 0 & 0 \end{bmatrix} \tag{6-23}$$

with:

$$\mathbf{v_1} = \begin{bmatrix} 0 \\ 0 \\ \frac{12\left(-c_2 g m_2 \cdot \left(12L_1c_2+L_2^2+12c_2^2\right)+\left(L_2^2+12c_2^2\right)\left(L_1 g m_2+c_1 g m_1+c_2 g m_2\right)\right)}{L_1^2L_2^2m_1+12L_1^2L_2^2m_2+12L_1^2c_2^2m_1+12L_2^2c_1^2m_1+144c_1^2c_2^2m_1} \\ \frac{12\left(c_2 g\left(L_1^2m_1+12L_1^2m_2+24L_1c_2m_2+L_2^2m_2+12c_1^2m_1+12c_2^2m_2\right)-\left(12L_1c_2+L_2^2+12c_2^2\right)\left(L_1 g m_2+c_1 g m_1+c_2 g m_2\right)\right)}{L_1^2L_2^2m_1+12L_1^2L_2^2m_2+12L_1^2c_2^2m_1+12L_2^2c_1^2m_1+144c_1^2c_2^2m_1} \end{bmatrix}$$

$$\mathbf{v_2} = \begin{bmatrix} 0 \\ 0 \\ \frac{12\left(c_2 g m_2\left(L_2^2+12c_2^2\right)-c_2 g m_2 \cdot \left(12L_1c_2+L_2^2+12c_2^2\right)\right)}{L_1^2L_2^2m_1+12L_1^2L_2^2m_2+12L_1^2c_2^2m_1+12L_2^2c_1^2m_1+144c_1^2c_2^2m_1} \\ \frac{12\left(-c_2 g m_2 \cdot \left(12L_1c_2+L_2^2+12c_2^2\right)+c_2 g\left(L_1^2m_1+12L_1^2m_2+24L_1c_2m_2+L_2^2m_2+12c_1^2m_1+12c_2^2m_2\right)\right)}{L_1^2L_2^2m_1+12L_1^2L_2^2m_2+12L_1^2c_2^2m_1+12L_2^2c_1^2m_1+144c_1^2c_2^2m_1} \end{bmatrix} \tag{6-24}$$

and we obtain

$$B := \left.\frac{\partial f}{\partial u}\right|_{\substack{x=\mathbf{0}\\u=0}} = \begin{bmatrix} 0 \\ 0 \\ \frac{12\left(L_2^2+12c_2^2\right)}{L_1^2L_2^2m_1+12L_1^2L_2^2m_2+12L_1^2c_2^2m_1+12L_2^2c_1^2m_1+144c_1^2c_2^2m_1} \\ \frac{12\left(-12L_1c_2-L_2^2-12c_2^2\right)}{L_1^2L_2^2m_1+12L_1^2L_2^2m_2+12L_1^2c_2^2m_1+12L_2^2c_1^2m_1+144c_1^2c_2^2m_1} \end{bmatrix} \tag{6-25}$$

And again we can convert these continuous matrices A_c and B_c to discrete time with a zero-order hold mechanism:

$$\begin{aligned} A_d &= e^{A_c h} \\ B_d &= A_c^{-1}\left(e^{A_c h} - I\right) B_c \end{aligned} \tag{6-26}$$

Chapter 7

Results

With the different dynamical systems and their controllers, we can set up different experiments. In this chapter, we will vary different factors in different neural networks to see how they influence their performance. For which we use two different metrics. We start by comparing different activation functions. Next, we investigate different sampling strategies and we implement max-plus and min-plus layers. We also compare different activation functions on our double pendulum system.

7-1 Activation functions

Our goal is to find if there are differences between the activation functions, therefore we trained 11 different neural networks with the same structure but different activation functions. Each network has 3 hidden layers with 8 neurons each fully connected. The output layer consists of 1 neuron with linear activation. Section 5-2-2 discusses the procedure further. We will use two metrics to judge the performance, the validation loss after training and a custom performance metric.

7-1-1 Controller for linearized system

For the linearized system, we expect a PWA control law, which is what we observe. Figure 7-1 shows both the original dataset obtained from the linearized MPC controller labeled "Linearized MPC" and the approximations made. The area shaded in gray marks where there is no data in the dataset because there was no solution found, or computation took too long. It shows the control input u the controller gives at that point in state space. For this dataset and the following PWA datasets, we used a dense uniform grid sampling method with 151 samples in each dimension, resulting in $151 \times 151 = 22801$ samples total. The original MPC controller allows for constraint satisfaction, however, the approximations do not guarantee these (physical) constraints stay satisfied so we impose them on the output of the neural

network $\hat{u}_{nn}$ to obtain the input we apply u_{nn}:

$$u_{nn} = min(max(\hat{u}_{nn}, -10), 10).$$

We can observe from the control laws that all the used activation functions seem to get a decent approximation of the original MPC controller. From this figure however, we can not observe much difference, for this, we use our other two metrics that are displayed in Figure 7-2 and 7-3

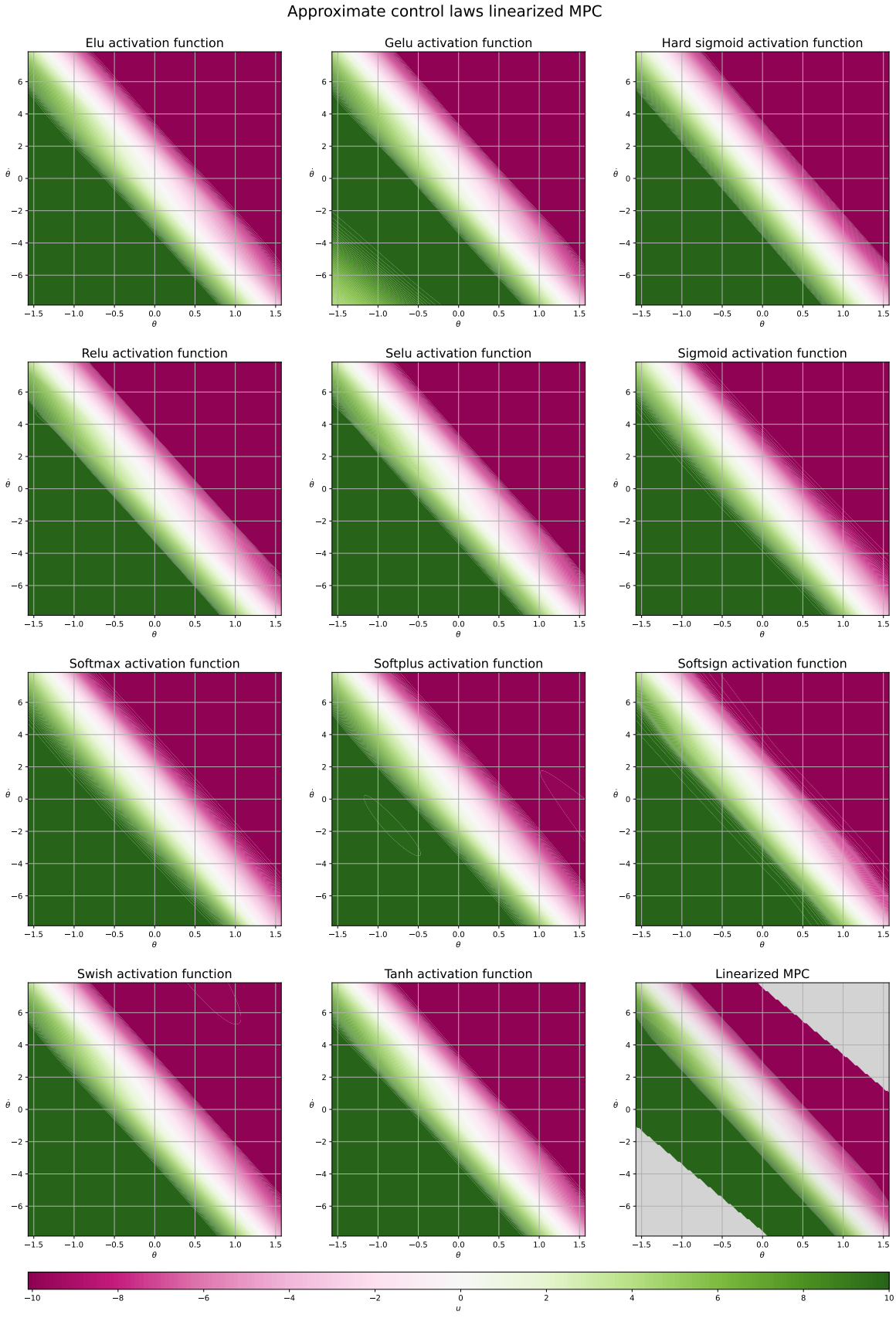
Approximate control laws linearized MPC
Elu activation function
Gelu activation function
Hard sigmoid activation function
Relu activation function
Selu activation function
Sigmoid activation function
Softmax activation function
Softplus activation function
Softsign activation function
Swish activation function
Tanh activation function
Linearized MPC
θ
θ̇
u

Figure 7-1

Figure 7-2 displays the loss of the trained networks and the validation loss. We observe some more differences here. The Relu activation function has the lowest loss and validation loss, which is remarkable since it is one of the most commonly used activation functions in neural networks. The other MMPS activation function, the Hard sigmoid, has the highest loss, which is unexpected since it could theoretically exactly represent a PWA function.

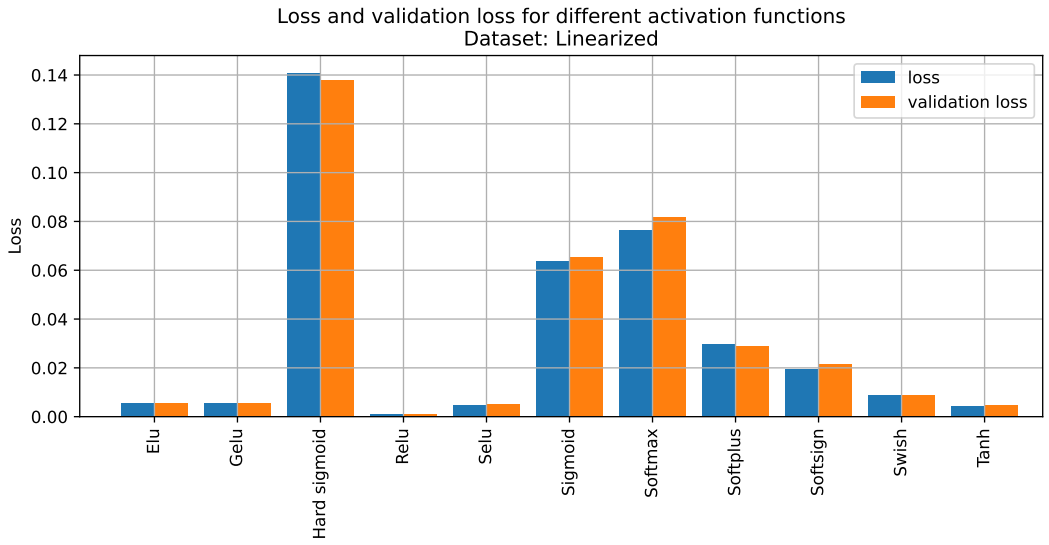

Figure 7-2

Figure 7-3 shows the cost of a set of 150 trajectories from a set of initial states. The benchmark is here the linearized MPC controller executing its policy for 100 timesteps. The cost function is similar to the MPC controller objective function, with the total cost, state cost, and input cost as:

$$\begin{aligned} V_{\text{total}}\left((x(1:150), u(1:150)\right) &= \frac{1}{2}\sum_{k=1}^{150}\left[V_{\text{state}}\left(x(k)\right) + V_{\text{input}}\left(u(k)\right)\right] \\ V_{\text{state}}\left(x(k)\right) &= x(k)^T Q x(k) \\ V_{\text{input}}\left(u(k)\right) &= u(k)^T R u(k) \end{aligned} \tag{7-1}$$

The cost is normalized using the benchmark cost and centered around 0. This way we can analyze the relative state and input cost as well as the total cost.

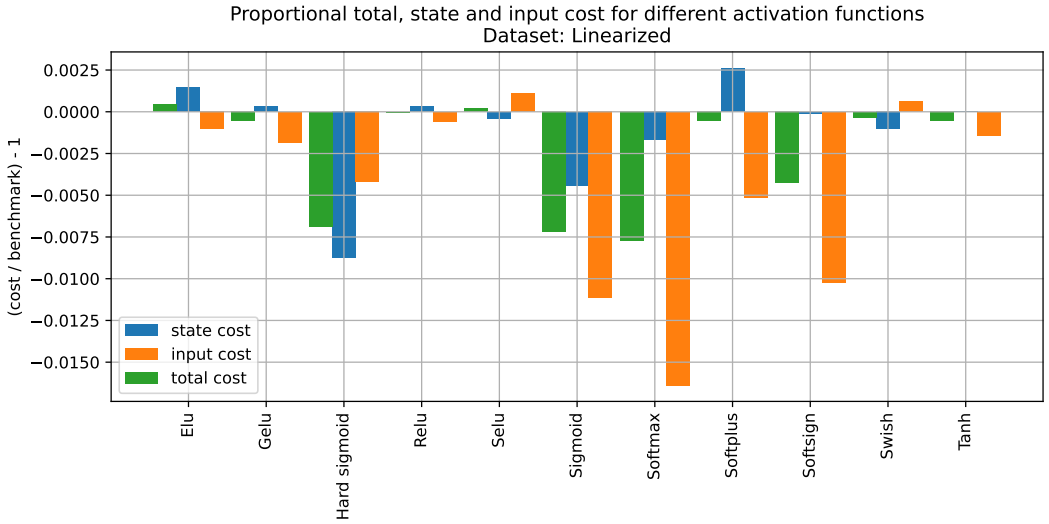

Figure 7-3

What we observe is that some controllers perform even better than the MPC controller. This seems incorrect, but the MPC controller is for a linear system and the simulation is done on a nonlinear system. So the linearized MPC control law is not optimal for this nonlinear system. By deviating from this control law, which happens when approximating, the result might be a lower cost. This metric means that the closer the score is to be exactly zero, the closer the approximate neural network controller is to the MPC controller. We see that the networks with the lowest validation loss in Figure 7-2, the Elu, Gelu, Relu, and Tanh networks, are also closest in cost to the benchmark control law. The networks with the highest validation loss, Hard sigmoid, Sigmoid, and Softmax, all have a lower cost when compared to the benchmark. For the Sigmoid and Softmax, there seems to be a larger difference in the input cost, meaning this controller gives less input than the linearized MPC controller.

7-1-2 Controller PWA system 1-norm

Now we want to compare how well we can approximate a different MPC control law. For this, we take a controller of the PWA system described in Section 6-2. The optimal control strategy for the MPC controller is described in Section 3-4-2, with a 1-norm cost. What we would expect here is also a PWA control law, since Theorem 3.8 states that a polyhedral PWA solution exists. The solver we use, however, through sampling yields a solution that does not seem to be polyhedral PWA. The solver also takes much longer than the linearized MPC controller to find a solution. It takes around 3.2 seconds per point on average but with some points taking much longer. Therefore a timeout is set to 10 seconds of computation, which gives some suboptimal solutions. Looking at Figure 7-4 we can see that some points in the dataset labeled "1-norm MPC" mostly in the areas where the control law is nearly constant are not optimal. The gray marked areas show where there is no data in the dataset. For the approximations, we see a lot more diversity in the shapes of the functions. We see that MMPS activation functions yield a result with straight lines, whereas other functions such as the tanh give a more smooth function. To see how well these approximations perform and how well the approximations fit we need to look at our other metrics.

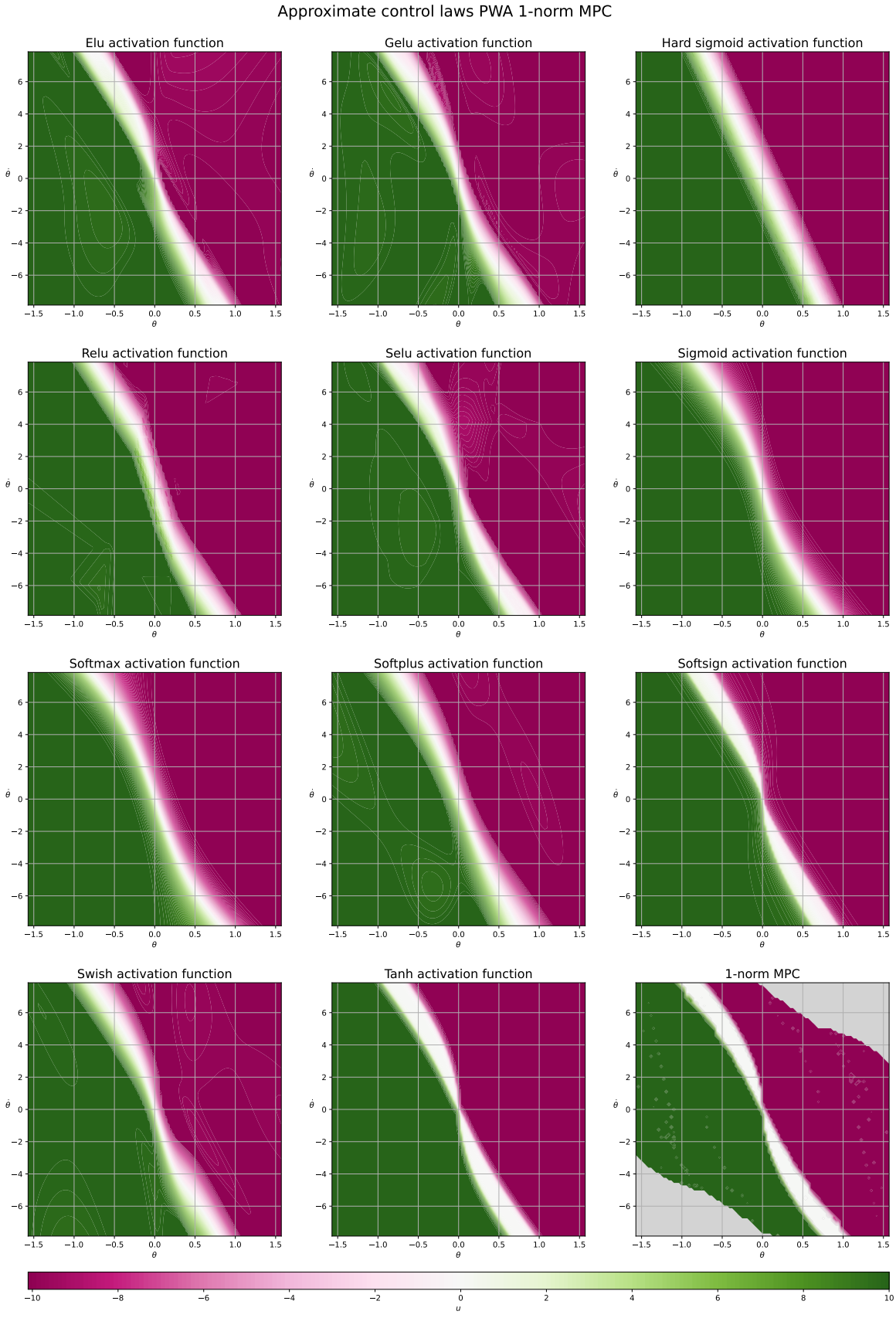
Approximate control laws PWA 1-norm MPC
Elu activation function
Gelu activation function
Hard sigmoid activation function
Relu activation function
Selu activation function
Sigmoid activation function
Softmax activation function
Softplus activation function
Softsign activation function
Swish activation function
Tanh activation function
1-norm MPC
θ
θ̇
u

Figure 7-4

Figure 7-5 shows the resulting loss and validation loss of the approximate control laws. What immediately becomes clear is that almost every loss is much higher than the loss of the approximations of the linearized MPC controller. Only the tanh function has a comparable validation loss. looking at Figure 7-4 we would expect the Hard sigmoid to have the highest loss since the approximation does not seem to follow the same shape as the dataset. This is indeed the case. The other MMPS activation function Relu seems to perform average compared to other smooth functions.

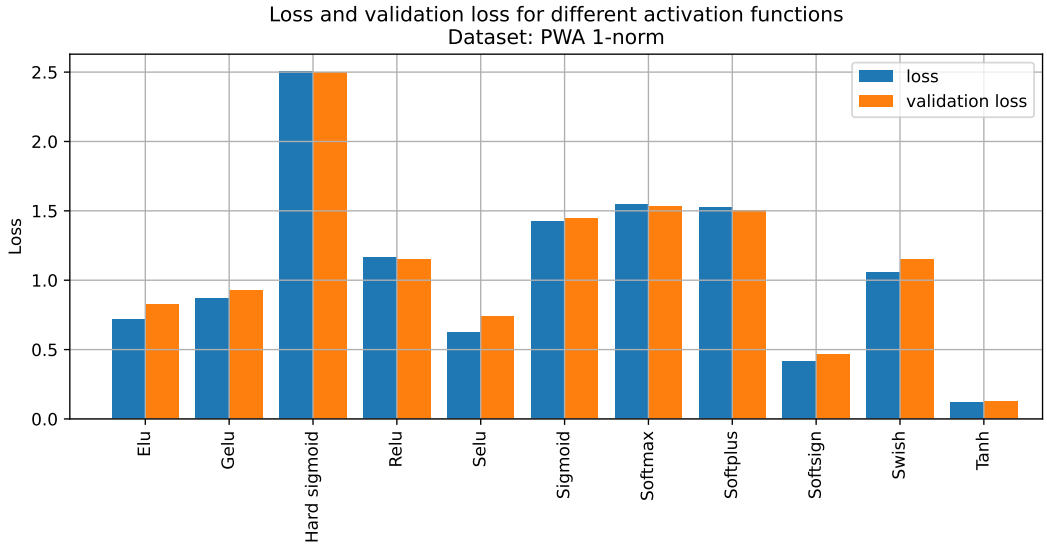

Figure 7-5

Now to analyze the comparative cost of the networks we use a metric that is similar to the corresponding MPC optimization cost, which is largely the same but uses a different norm:

$$\begin{aligned} V_{\text{total}}\left((x(1:150), u(1:150)\right) &= \frac{1}{2}\sum_{k=1}^{150}\left[V_{\text{state}}\left(x(k)\right) + V_{\text{input}}\left(u(k)\right)\right] \\ V_{\text{state}}\left(x(k)\right) &= ||Qx(k)||_1 \\ V_{\text{input}}\left(u(k)\right) &= ||Ru(k)||_1 \end{aligned} \tag{7-2}$$

We see the result in Figure 7-6, here we see that some of the networks with low validation loss have a lower cost. This is the case for the Softsign and Tanh activation networks. What does seem to differ is the performance of the Selu activation network, while having a relatively low loss as shown in Figure 7-5, its relative cost is worse compared to the Hard sigmoid, Sigmoid, Softmax, and Softplus, which all have a higher validation loss. This seems to indicate that having a more precise approximation of the control law for the PWA approximation of the nonlinear system does not necessarily guarantee better performance in a nonlinear simulation.

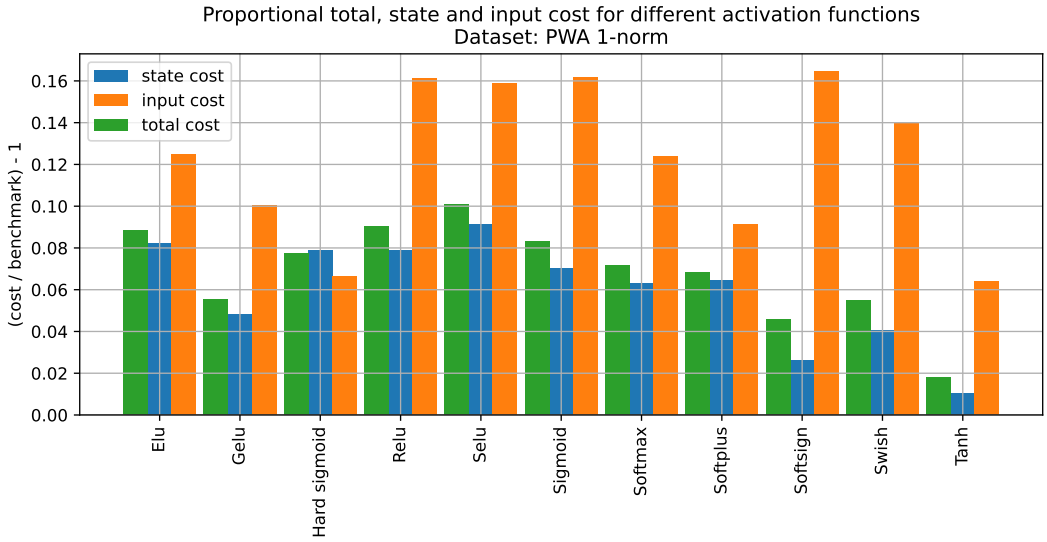

Figure 7-6

7-1-3 Controller PWA system infinity-norm

This controller is analogous to the PWA system 1-norm controller but with the ∞-norm instead of the 1-norm. The resulting control law, displayed in Figure 7-7, is also similar but has a larger region where the control input u is zero. Despite the control laws looking very similar, the approximations differ from the 1-norm controller. The hard sigmoid looks more similar to the Relu activation network and other activation function networks such as the Softplus, have a different shape from what they had in the 1-norm case.

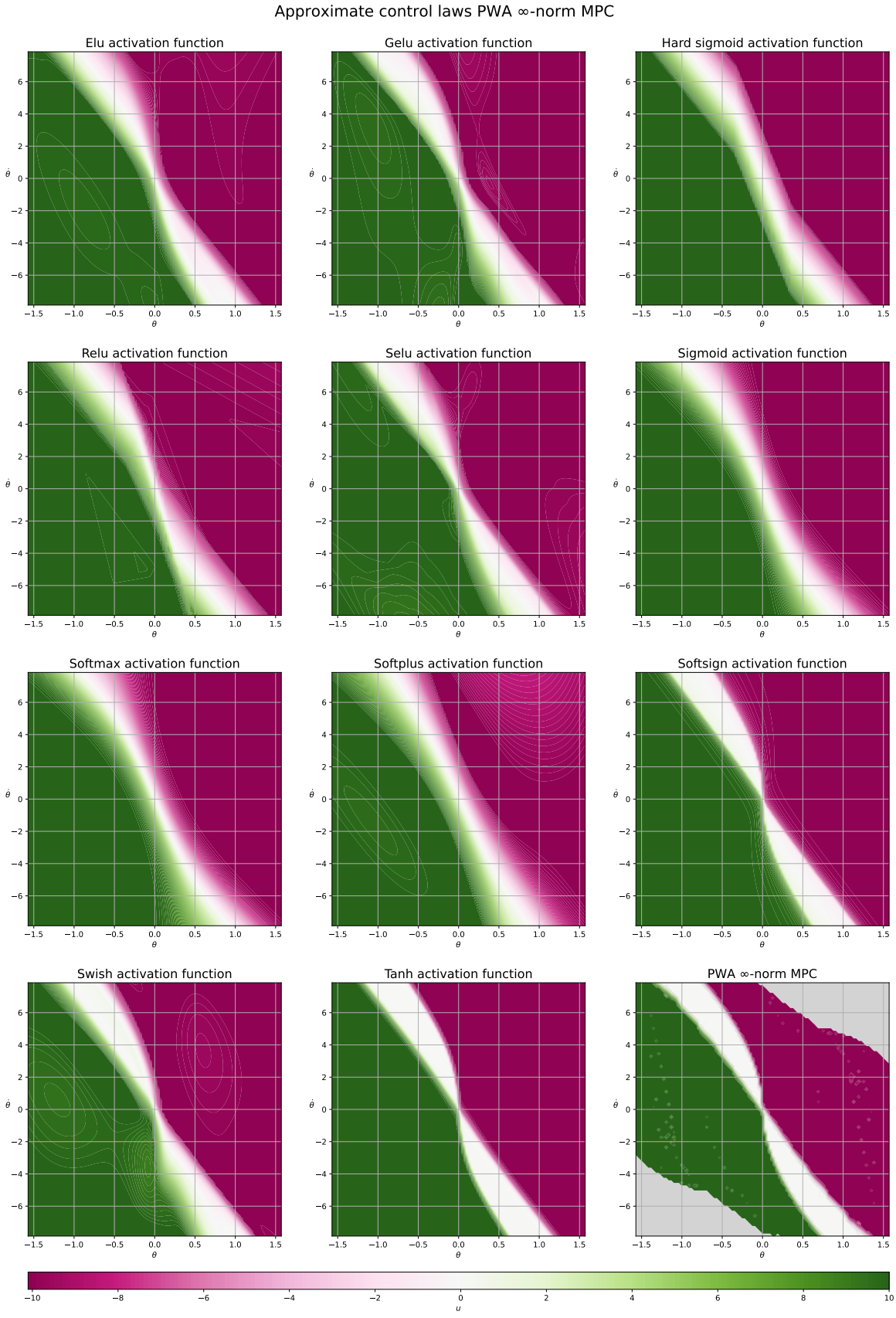
Approximate control laws PWA ∞-norm MPC
Elu activation function
Gelu activation function
Hard sigmoid activation function
Relu activation function
Selu activation function
Sigmoid activation function
Softmax activation function
Softplus activation function
Softsign activation function
Swish activation function
Tanh activation function
PWA ∞-norm MPC
θ
θ̇
u

Figure 7-7

Looking at the loss and validation loss of the networks in Figure 7-8 we see that the loss for a lot of networks is higher than the 1-norm and the linearized controller. Except for the Softsign and Tanh activation networks. The Sigmoid, Softmax, and Softplus also had a high loss for the 1-norm controller, but here they have even higher losses.

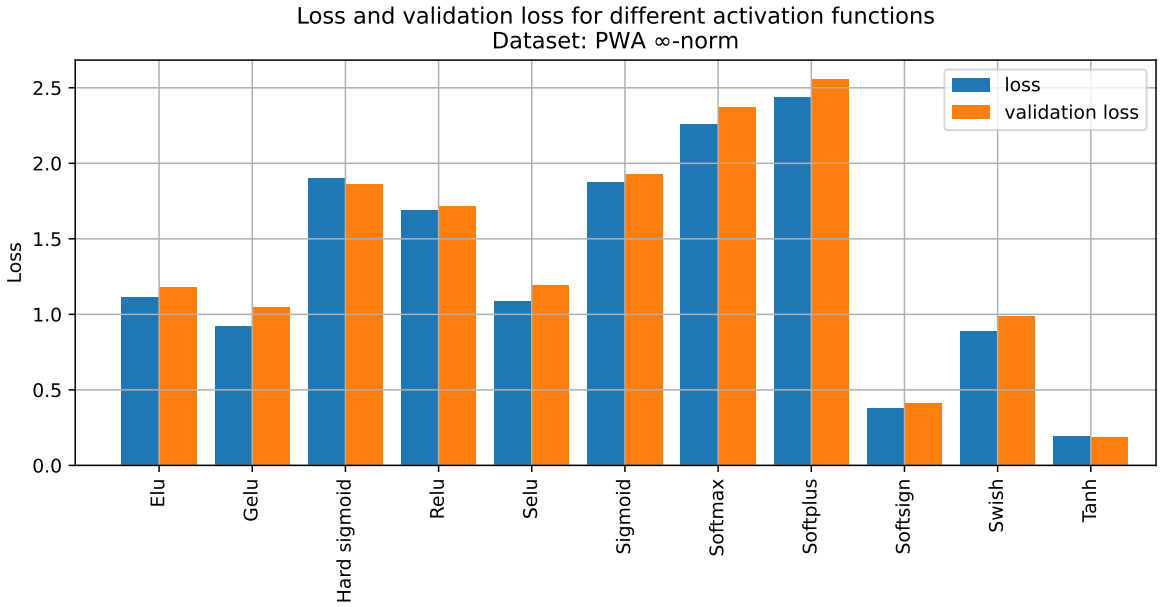

Figure 7-8

Comparing the losses from Figure 7-8 to the relative cost in Figure 7-9 we see that some networks have a proportionally lower input cost, meaning they give smaller inputs than the MPC controller. We also observe that the Softsign and Tanh with the lowest losses also have the lowest costs, but the activation networks with the highest losses do not necessarily have the highest cost. The Softmax for example has a high validation loss but scores relatively well here.

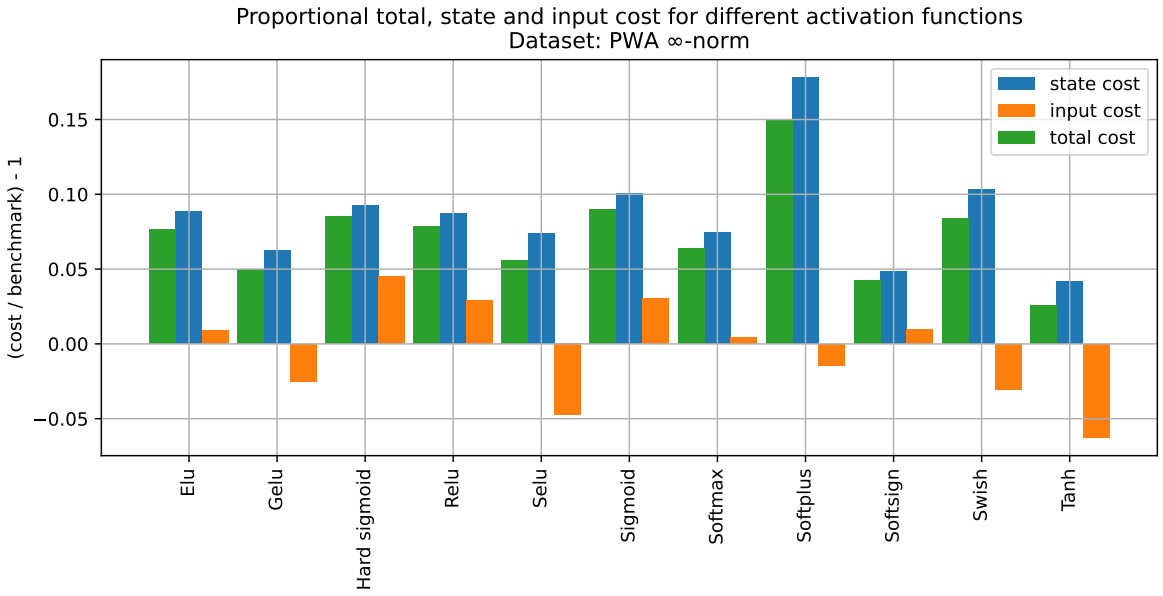

Figure 7-9

7-2 Sampling Strategy

Since we want to approximate an MPC controller from data, we use sampling. In the previous experiment, we had a dense uniform grid of samples, but in reality, so many samples are hard to obtain. This also requires exponentially more samples if the dimensionality of the system increases. A grid of 151×151 samples would require a lot of sampling. That is why we want to compare 3 different sampling strategies that use far fewer sampling points. as described in Section 5-2-1.

- A random sampling method with a uniform probability of sampling a point (Random sampling)
- A stratified sampling method with a lower probability of sampling a point in general, except if a point varies a lot from its neighbors (Stratified sampling)
- A more sparse uniform grid sampling method (Uniform sampling)

Figure 7-10 shows the datasets and the resulting approximate control laws from different activation function networks. The gray shaded areas in the datasets are spots where there is no data. We obtained these datasets from the PWA 1-norm controller. We observe quite a few differences between the shapes of the control laws. For the random sampling method, there is hardly any swirling effect visible on the approximate control laws. The effect is mostly visible in the results from the stratified sampling method. The tanh function networks, even a bit in the random sampling and uniform sampling method, seem to capture this shape the most. What also stands out is the lighter shade in the uniform sampling method sigmoid network. We will analyze them further with our other metrics.

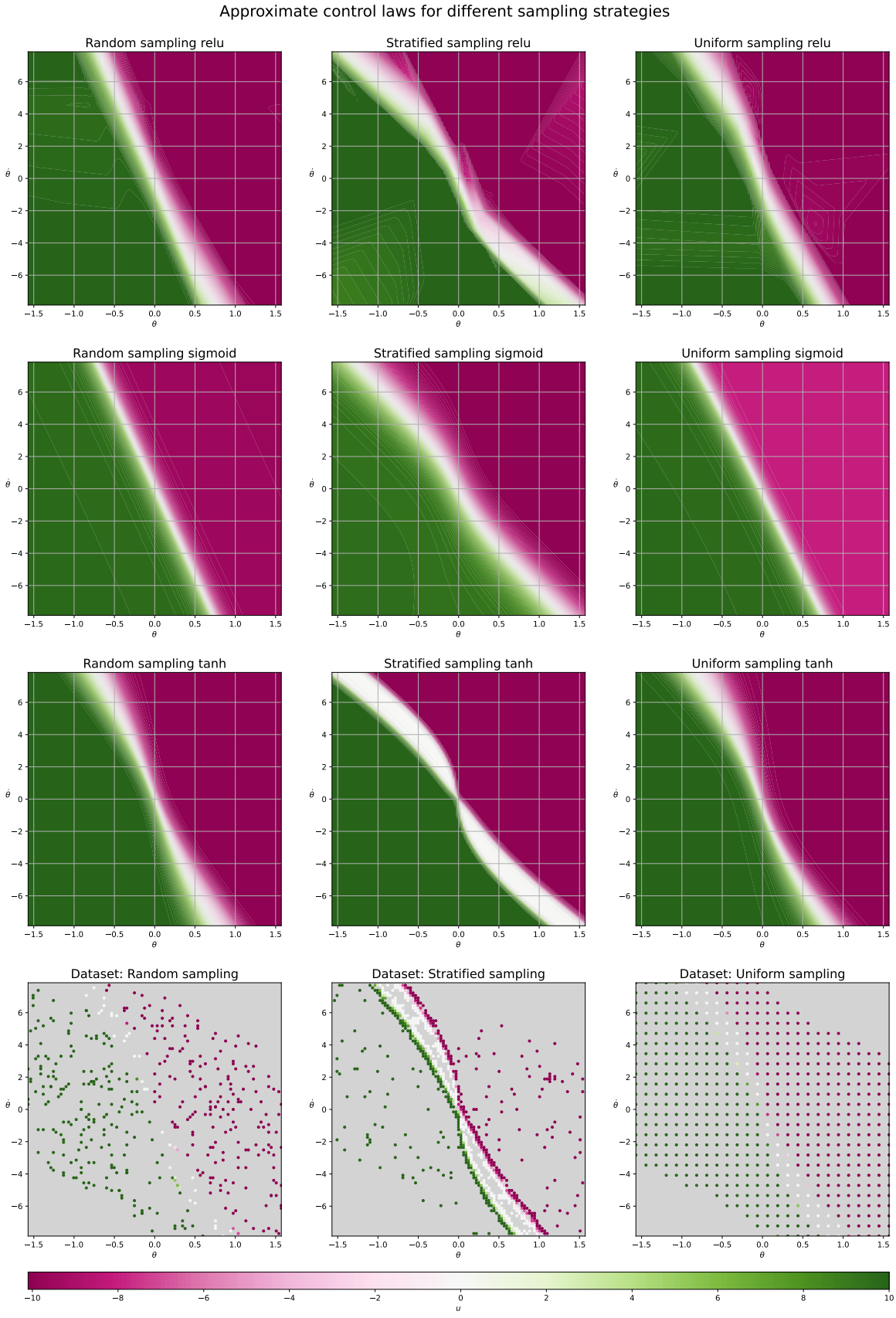
Approximate control laws for different sampling strategies
Random sampling relu
Stratified sampling relu
Uniform sampling relu
Random sampling sigmoid
Stratified sampling sigmoid
Uniform sampling sigmoid
Random sampling tanh
Stratified sampling tanh
Uniform sampling tanh
Dataset: Random sampling
Dataset: Stratified sampling
Dataset: Uniform sampling
θ
θ̇
u

Figure 7-10

Looking at the losses in figure 7-11, we see overall larger losses than our previous dense uniform sampling method. The largest losses seem to be using the sigmoid activation function networks. The tanh activation networks seem to have good losses with these fewer-data methods. We can analyze how well they perform with our other metric

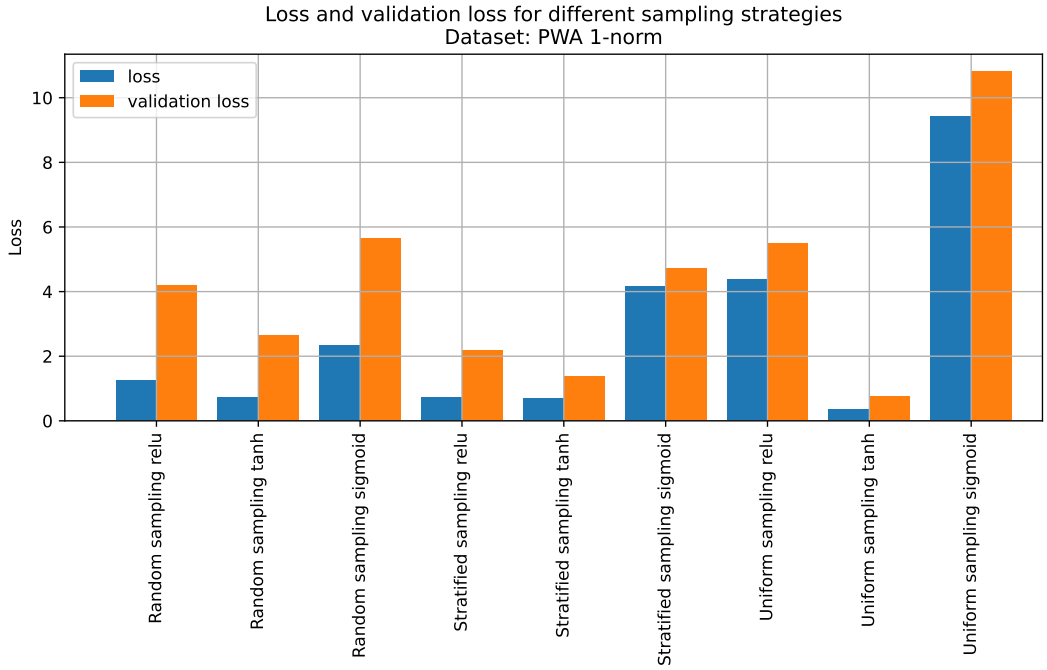

Figure 7-11

We use again the same metric as the other PWA 1-norm controller as described in Equation 7-2. The resulting normalized and centered state, input, and total cost are displayed in Figure 7-12 Here we see the biggest differences between our sampling strategies. Here the uniform sampling strategies seem to perform nearly the same as our benchmark with only a slightly higher total cost. The stratified sampling deviates slightly more from the benchmark but still outperforms the random sampling method. The input cost is also comparatively lower than our benchmark using the random and stratified sampling strategies. They seem to structurally give a smaller input than the benchmark. Perhaps the resulting networks have a larger region where they give zero input compared to the benchmark and the sparse uniform sampling method.

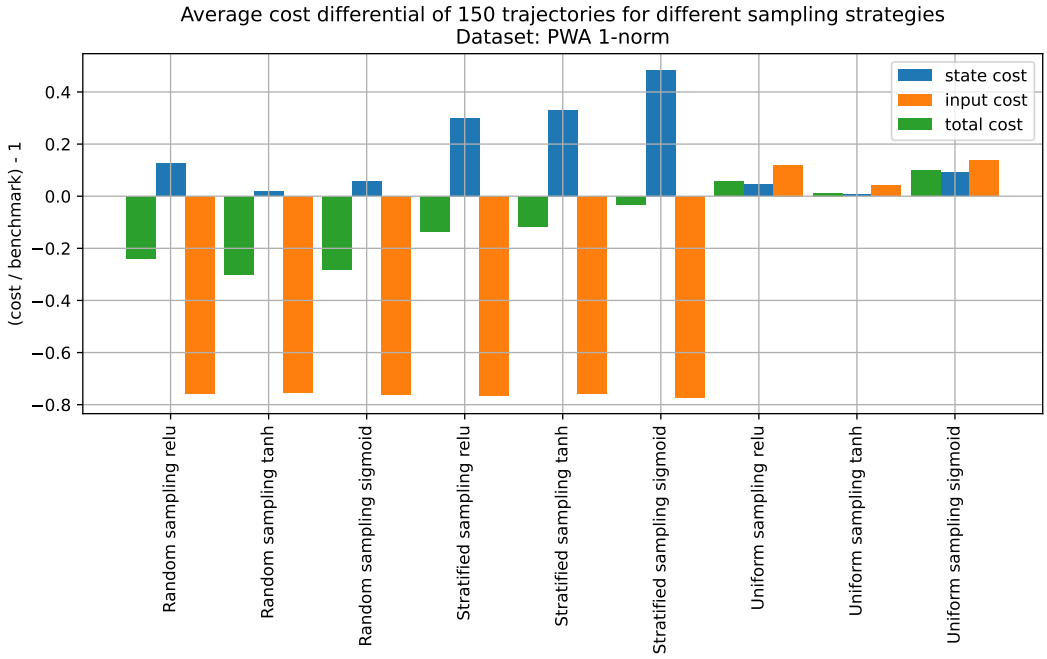

Figure 7-12

7-3 Maxplus and Minplus layers

Another type of network that we want to explore is neural networks with maxplus and minplus layers as described in Section 4-1-2. We will consider three network structures. All networks have 7 hidden layers, of which the first is a layer with linear activation and 1 output layer with linear activation. The 6 other hidden layers are either

- All max-plus layers
- All Relu layers (with similar results to Section 7-1)
- Alternating max-plus and min-plus layers
- Aternating max-plus and Relu layers

Each hidden activation layer will have 8 neurons each, and the first linear layer has 16 neurons so that every network has the same number of parameters. We will look at how well these networks approximate our previously mentioned controllers. The result is shown in Figure 7-13 and we can immediately see that there are some questionable control laws for the max-plus only solution for the PWA 1-norm and PWA ∞-norm controllers. The same goes for the max-plus and min-plus alternating networks, but here we see an interesting pattern of sharp angles forming. We have seen the results for a smaller Relu network in Section 7-1 and only the max-plus and Relu alternating networks and the maxplus-only approximation for the linearized dataset come close to this, .

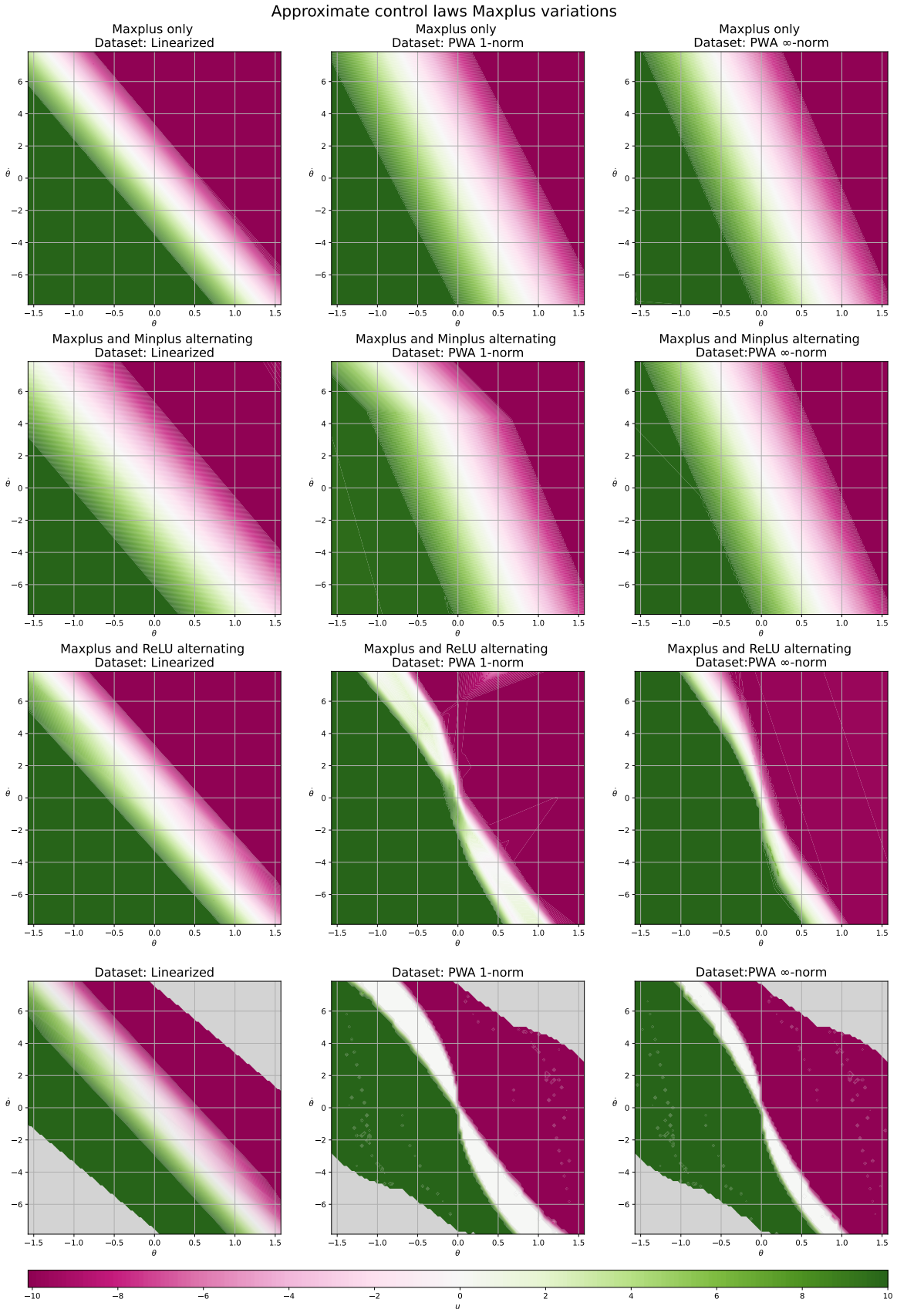
Approximate control laws Maxplus variations
Maxplus only
Dataset: Linearized
Maxplus only
Dataset: PWA 1-norm
Maxplus only
Dataset: PWA ∞-norm
Maxplus and Minplus alternating
Dataset: Linearized
Maxplus and Minplus alternating
Dataset: PWA 1-norm
Maxplus and Minplus alternating
Dataset:PWA ∞-norm
Maxplus and ReLU alternating
Dataset: Linearized
Maxplus and ReLU alternating
Dataset: PWA 1-norm
Maxplus and ReLU alternating
Dataset:PWA ∞-norm
Dataset: Linearized
Dataset: PWA 1-norm
Dataset:PWA ∞-norm
θ
θ̇
u

Figure 7-13

Based on the shapes of the approximations in Figure 7-13 we would expect high losses for the max-plus and min-plus alternating networks, which is what we observe in Figure 7-14. These networks do perform better on the linearized dataset, but still, the networks with Relu only or Relu and max-plus alternating networks have far lower losses. The max-plus only network seems to perform well on the linearized dataset, only slightly higher than the Relu only and the Relu and max-plus alternating network, but only on the Linearized datset. The Linearized controller dataset is PWA, and the PWA controller dataset is not, which might be the cause of this larger difference. We also see that for the Linearized dataset, the max-plus and Relu alternating network has nearly the same loss as the Relu-only network, whereas, for the PWA 1-norm and ∞-norm dataset, the Relu only has a lower loss. We will see if this holds in our other metric.

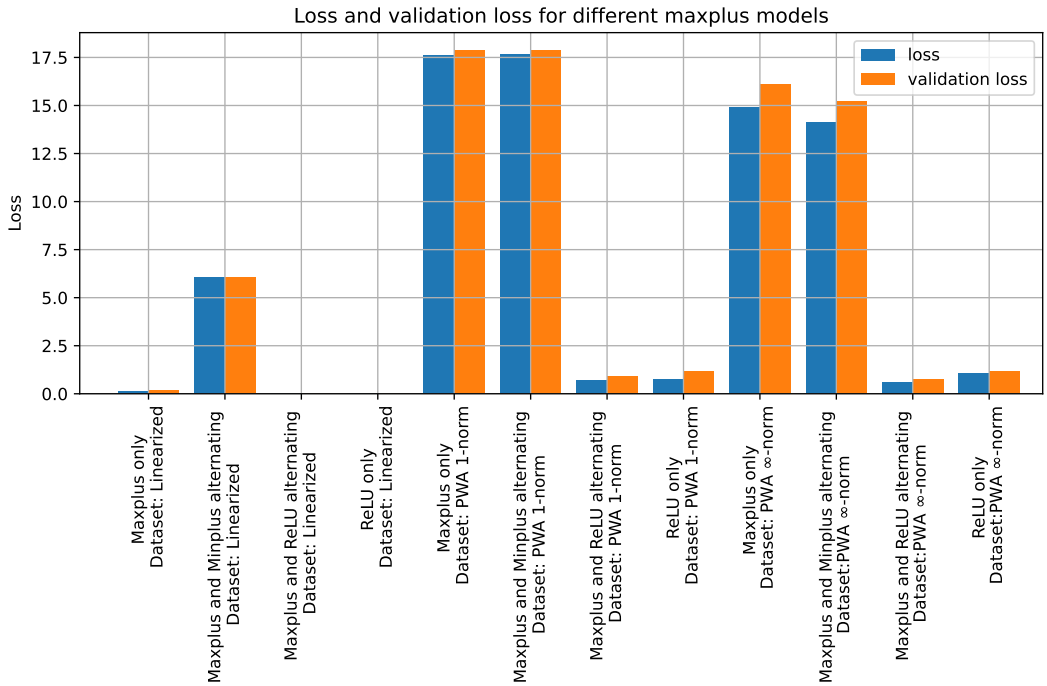

Figure 7-14

Figure 7-15 shows the custom score metric for all networks except the networks that did not seem to converge, The max-plus only PWA 1-norm and infinity norm networks. They are excluded because simulating them on the nonlinear system would yield unstable results, giving a large score that is not comparable to the other control law results. We observe from the figure that the linearized dataset has a result that is very similar to the benchmark, the max-plus and Relu alternating network is nearly the same. The max-plus and min-plus alternating networks on both controllers have a high cost, which we would expect since they deviate quite a bit from their respective datasets. The max-plus and Relu alternating networks still have a relatively low cost when approximating these PWA controller datasets.

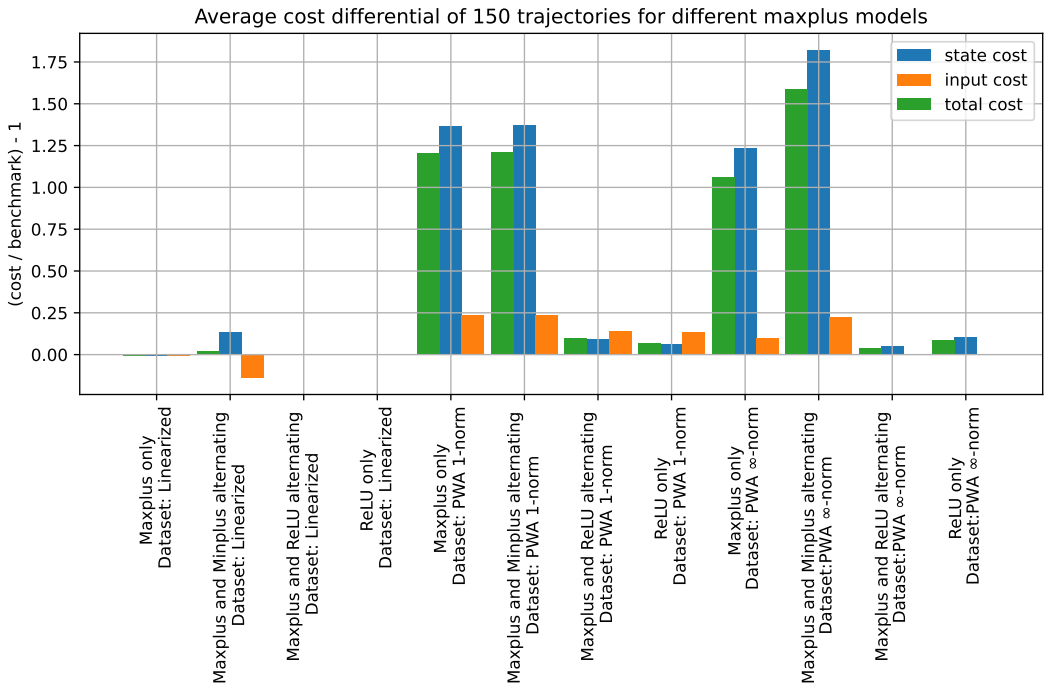

Figure 7-15

7-4 Double Pendulum

Now to consider another system, a double inverted pendulum as described in Section 6-3-1. We expect the control law to be polyhedral PWA, though this is hard to visualize since the resulting state space is four-dimensional. Therefore we will only be using the loss and cost metrics to analyze the approximations. To get a dataset here we will sample from simulated trajectories with initial stationary rod positions. So we select a set of initial rod configurations and then run a simulation. If the simulation is stable, its states and the input the controller gives are added to the training data. The resulting losses are shown in Figure 7-16. They appear to be quite low, similar to the losses of the single pendulum linearized MPC controller system. The lowest validation loss is for the Hard sigmoid network, which is an MMPS function. The other MMPS function Relu seems to have a bit higher loss. The Softmax has the highest losses, but comparatively, all the losses are quite close. The validation losses are for some networks much higher than the (training) losses. We also see the max-plus network, with 1 starting linear layer and for the rest max-plus layers have a higher validation loss.

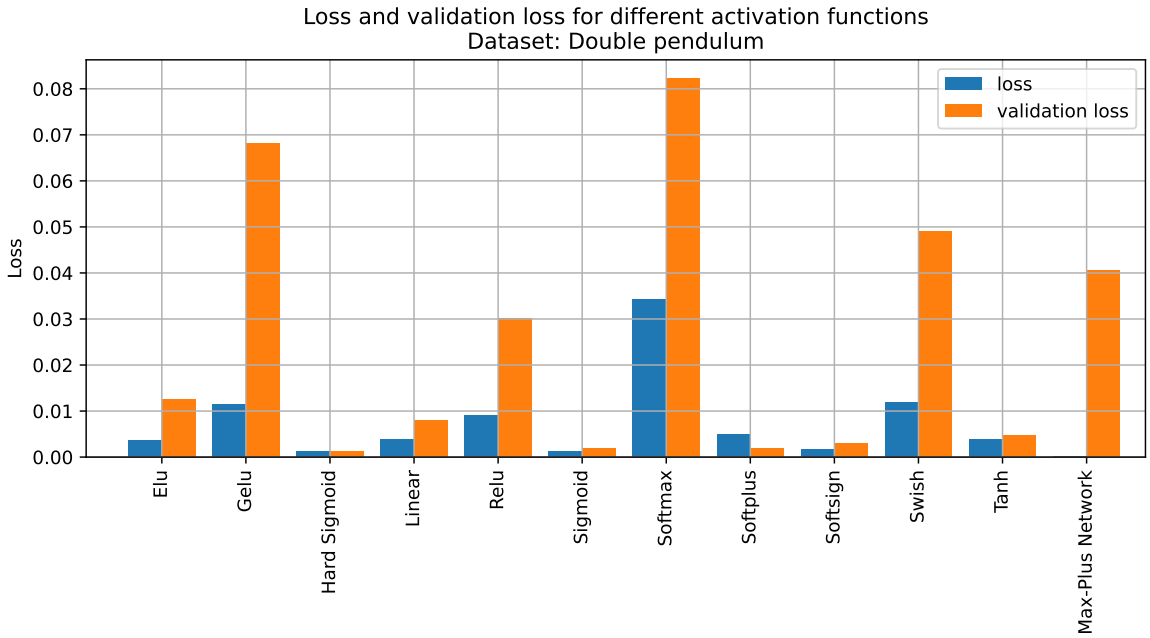

Figure 7-16

Figure 7-17 shows that also the proportional cost metric is very similar for most activation function networks. The Relu activation network stands out here since it is the only network with a higher cost than the benchmark and by a significant margin compared to the other networks. Even though other networks have higher losses, they still have scores that are closer to the benchmark. We do see here that the max-plus network has the lowest total cost, but it is similar to many other networks.

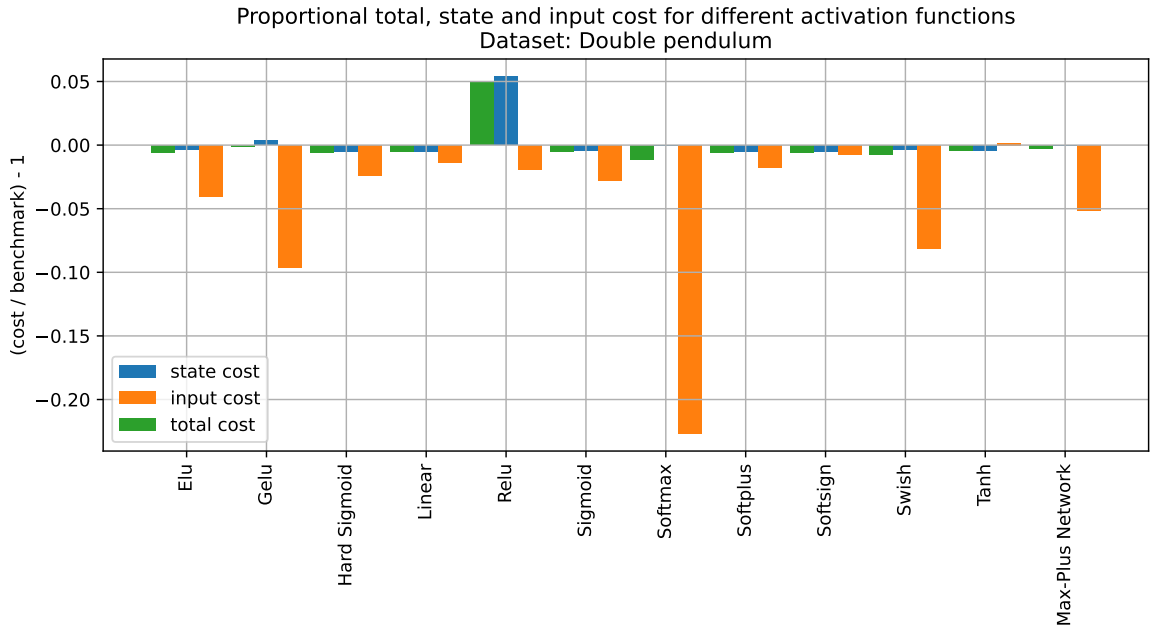
Proportional total, state and input cost for different activation functions
Dataset: Double pendulum
(cost / benchmark) - 1
0.05
0.00
−0.05
−0.10
−0.15
−0.20
state cost
input cost
total cost
Elu
Gelu
Hard Sigmoid
Linear
Relu
Sigmoid
Softmax
Softplus
Softsign
Swish
Tanh
Max-Plus Network

Figure 7-17

Chapter 8

Conclusion

8-1 Conclusion to experiments

In Section 1-2-1 we posed a set of research questions that we can now answer based on the results of our experiment.

First, as pointed out in nearly all sections of Chapter 7, there are significant differences when approximating a PWA control law compared to a non-PWA control law. Our PWA control law arose from a linearized MPC controller and is used throughout the experiments with different sampling activation functions and max-plus and min-plus layers. For all these experiments the PWA-control law saw a lower validation loss and they score closer to the benchmark on the performance metric than the experiments with our non-PWA control laws.

The conclusion we can draw from the first experiment with different activation functions is that there is definitely merit to using different activation functions for different control laws. Though Section 7-1-1 shows the fact that an activation function is an MMPS function does not automatically guarantee better performance when approximating a PWA control law. The Relu activation function does perform the best here, but the other MMPS activation function, the hard sigmoid function performs way worse. Looking at the other two control laws that are non-PWA and the performance there, some non-MMPS activation functions have a better performance than the most commonly used Relu function. We also see this in our double pendulum system where the Relu function scores the worst in the custom performance metric compared to other activation functions. We did expect this somewhat for non-PWA functions, as only MMPS approximations could yield a worse approximation. However, also for PWA functions with small linear regions, a non-perfect MMPS approximation may be outperformed by a smooth approximation function, which happened in our experiments.

We also looked into different sampling strategies, and these have a large influence on the performance of the approximating neural networks. For this, we used a non-PWA dataset and observed the performance of different activation function neural networks. What was interesting to see here was the difference between the validation loss and other performance metric. The stratified sampling strategy achieved some of the lowest validation losses, but the

uniform sampling network was performing more closely to the benchmark. This means that not only the areas where points vary more from their neighbors are important to capture, but also the more "simple" areas. We also learned that a uniform sampling strategy is not always viable, with our double pendulum system, where we solve this with yet another sampling strategy by simulating state trajectories over time and using them as our training data.

The more novel max-plus and min-plus layers were also implemented and we analyzed the performance in Section 7-3. We see that or the max-plus and min-plus alternating networks offer a far worse performance compared to networks with Relu activations. The max-plus only network did offer a good performance comparable to other activation functions on the linearized dataset. On the double pendulum system, we presumably have another PWA control law and here it also performs comparatively to networks with a traditional structure. So a network that starts with a linear layer, has more hidden max-plus layers and a final linear output layer can offer a good performance in approximating polyhedral PWA control laws. As for faster performance, the addition operation on most modern processors for floating point numbers is about the same number of clock cycles as floating point multiplication, as shown in Appendix D.3 in [59]. This means that with a low-level programming language, there will not be a significant difference in speed when only using max and plus, or min and plus operators, instead of also including multiplication.

8-2 Main conclusion

The goal of this thesis is to discover what various factors are of influence on the performance of neural network approximations of MPC control laws. One factor we are interested in, especially for piecewise affine control laws, is max-min-plus-scaling functions. These are functions that contain max, min, and scaling operations and are used to model discrete event systems. It is known that max-min-plus-scaling functions and piecewise affine functions are equivalent, and can be represented as one another. We also briefly touched on the related max-plus and min-plus algebra, which comes back later with max-plus and min-plus layers in neural networks.

Next, we explored Model Predictive Control, which gives us the control laws that we want to approximate. Model Predictive Control is a technique where we have a model of a system and use future predictions of that model to compute an optimal control sequence. We discuss MPC for linear systems and their stability. We also discuss model predictive control for Mixed Logical Dynamical systems, which is another way to represent a piecewise affine system. In all of these controllers still, an optimization problem has to be solved at each timestep. This optimization problem can be computationally hard, and as a solution to this there sometimes exist an explicit control law. Applying an explicit control law is much faster than solving an optimization problem, but explicit control laws are not always available, so approximations are still practical here.

These approximations make use of a technique for approximations of various functions: neural networks. We discuss the structure of neural networks and what parameters they have. We also survey what activation functions are widely used in neural networks, and find that with the use of some activation functions, the neural network works as a MMPS function. There are also some novel neural networks: min-max-plus-scaling neural networks, which make use

of max-plus layers and min-plus layers. We also dive into how neural networks train by using gradient descend and what different update rules for this are commonly used.

With the knowledge of these topics, we set out to answer the main question and subquestions of this research by setting up experiments. We first set up different MPC controllers for a single pendulum system, one linearized controller, and two PWA controllers. We also set up a more complicated linearized controller for a double pendulum system. From these controllers, we extract 4 large datasets as our control law to estimate with several neural networks. The resulting

We first start with our 3 datasets for the single pendulum system. Here we first train several neural networks with different activation functions. We then, like for the rest of our experiments, use the validation loss and a custom performance score as a metric for their performance. We see that the different activation functions have varying performance. Not all MMPS activation functions have a better performance on the PWA control law, the hard sigmoid activation scores worse than non-MMPS functions. The MMPS activation function Relu does perform the best for the PWA control law. There is also a large difference when estimating non-PWA control laws. The PWA control laws obtained from the linearized controller are more accurately estimated by every activation function neural network.

Next, we use subsets of our large datasets to test the difference between a random sampling approach, a stratified sampling approach with more samples where the surrounding points vary from each other, and a uniform grid sampling approach. The stratified sampling has a lower validation loss, but the uniform sampling strategy scores better with our custom metric. What we learn from this is that capturing the area with more differences between points is just as important as capturing more 'simple' areas. For our double pendulum system we find that the amount of data for a uniform sampling strategy grows exponentially and to solve this we simulate trajectories from a lower dimensional set of initial states and use these trajectories as data, which also gives workable results.

We also implemented the max-plus and min-plus layers in neural networks. Alternating max-plus and min-plus alternating networks score poorly in both metrics, but the alternating max-plus and Relu network could get a similar performance to a pure Relu network. We also found that a network that starts with a linear layer, has more hidden max-plus layers and a final linear output layer can offer a good performance in approximating polyhedral PWA control laws. It does however perform very poorly on approximating non-PWA control laws.

Eventually, we found various factors that influence the performance of neural network approximations of MPC control laws. There are significant differences between approximating PWA and non-PWA control laws. The MMPS structure of neural networks can sometimes get a better approximation, though this is not guaranteed and sometimes non-MMPS activation functions achieve better results. Without doing more experiments we can not establish a clear pattern when MMPS structures outperform non-MMPS structures. Some sampling strategies, such as stratified sampling or uniform grid sampling offer good performance in approximating control laws, but the best suited strategy depends on the availability of data. Finally we found that some max-plus networks can offer a similar performance when approximating PWA MPC control laws compared to Relu networks.

8-3 Further research

Various factors that influence the performance of approximations of MPC control laws have been researched by this thesis, yet more possible factors have arisen, as well as other areas where further research is required.

8-3-1 More experiments and different systems

We could observe there are differences when using MMPS structures in a neural network compared to non-MMPS structures. We could see that sometimes an MMPS activation function would have a closer approximation of a PWA control law, but other times smooth activation functions got a closer performance. We could not find a clear pattern when this happens with our two different systems. For this more experiments with different systems yielding more PWA control laws need to be done.

8-3-2 Speed of max-plus and min-plus layers with different architectures

The floating point operations for multiplication and addition are roughly the same on most CPUs, so max-plus multiplication is not significantly faster than normal multiplication. However, there is a more significant difference between integer multiplication and integer addition. Some works have already looked into integer arithmetic to train integer neural networks [60]. This could be extended to max-plus and min-plus layers in neural networks. The work could also look more into the inclusion of $-\infty$ and ∞ in the weights of the network, which currently only uses finite weights.

8-3-3 Post-processing methods on neural network approximations

This thesis has done only limited post-processing on the neural network approximations of the control laws. Only saturating the output to satisfy physical constraints. In training, we can see that in maximization or minimization operations there can be parameters that become so small or large that the operation itself becomes redundant. In a traditional neural network, if weight values go to zero and a neuron becomes redundant, neurons can be removed during training with a technique called dropout [61]. A similar technique could be used for training weights that go to positive or negative infinity in a neural network with min-plus or max-plus layers.

8-3-4 More general forms of neural networks

We showed in Section 4-1-3 that we can write a network with max-plus and min-plus layers and a final or initial linear layer in a more compact form. This could possibly be generalized to work for any neural network with MMPS activation functions such as Relu or Hard sigmoid. These matrices in this general form grow significantly in size, but how exactly depends on the neural network. We know that the depth of a neural network helps more with expressiveness than the width of a network. How this translates to the size of the matrices of a general form

is yet to be researched. There is also not much research on the number of linear regions of networks other than Relu networks. Section 4-2 discusses upper and lower boundaries for the number of linear regions of a Relu network, as well as a way to compute an exact number. These boundaries do not necessarily hold for networks with max-plus and min-plus layers, so more research here could prove useful for estimating the required size of a network with max-plus and min-plus layers.

8-3-5 Stability and constraint validation for neural network controllers

As shown in Chapter 3, there are proofs to show that a (nonlinear) MPC controller is stabilizing the system. For neural network controllers, some stability proofs have been researched, mostly for Relu and Tanh networks such as [62] and [63]. These proofs hold for some activation functions, but not yet all that we applied here for example. It is also not guaranteed that a neural network controller does not violate constraints without post-processing. This would require some constrained neural network, which is done by [64]. It could be beneficial to research if such a constrained neural network yields a better result than a non-constrained neural network with post-processing.

8-3-6 Other sampling strategies and parallels to identification techniques

In this thesis, we used a dense uniform grid to get subsets to test sampling strategies. From this, we compared a sparse uniform grid sampling method, a random sampling method, and a stratified sampling method. More types of sampling are possible, for example, cluster sampling, or a different multi-stage sampling approach. We also used the trajectories from various initial states as data with our double pendulum system. This is closer to other system identification techniques. There has been some work done on input signal design for identification of max-plus systems [65]. More research can be done to compare this input signal design for max-plus systems to that for neural network approximations of PWA functions and non-PWA functions.

8-3-7 Other metrics to measure performance of approximation

We used how closely a control law is approximated as a measure of performance, but there could be other metrics. One flaw in the custom metric that we used is that trajectories further from the origin give a higher cost, meaning you can have a very close approximation near the origin, but if you have a slight error further away from the origin, the cost could come out to be quite high. What could also be a metric is the amount of data it costs to get a close enough approximation, or how many parameters are used in a network.

Appendix A

Appendices

A-1 Discretization of continuous time piecewise affine system

Consider a continuous time piecewise affine system in the form

$$\dot{x}(t) = Ax(t) + B_1 + B_2 u(t) \tag{A-1}$$

First solve for $x(t)$ and obtain the following:

$$\begin{aligned}
&\dot{x}(t) - Ax(t) = B_1 + B_2 u(t) \\
&e^{-At}\dot{x}(t) - e^{-At}Ax(t) = e^{-At}B_1 + e^{-At}B_2 u(t) \\
&\frac{d}{dt}\left(e^{-At}x(t)\right) = e^{-At}B_1 + e^{-At}B_2 u(t) \\
&e^{-At}x(t) - e^{0}x(0) = \int_0^t e^{-A\tau}B_1 + e^{-A\tau}B_2 u(\tau)d\tau \\
&e^{-At}x(t) = \int_0^t e^{-A\tau}B_1 + e^{-A\tau}B_2 u(\tau)d\tau + x(0) \\
&e^{At}e^{-At}x(t) = e^{At}\int_0^t e^{-A\tau}B_1 + e^{-A\tau}B_2 u(\tau)d\tau + e^{At}x(0) \\
&x(t) = e^{At}\int_0^t e^{-A\tau}B_1 + e^{-A\tau}B_2 u(\tau)d\tau + e^{At}x(0)
\end{aligned} \tag{A-2}$$

Next up denote $x(k)$ with timestep h as

$$\begin{aligned}
&x(k) = x(kh) \\
&x(k) = e^{Akh}x(0) + e^{Akh}\int_0^{kh} e^{-A\tau}B_1 + e^{-A\tau}B_2 u(\tau)d\tau \\
&x(k+1) = e^{A(k+1)h}x(0) + e^{A(k+1)h}\int_0^{(k+1)kh} e^{-A\tau}B_1 + e^{-A\tau}B_2 u(\tau)d\tau
\end{aligned} \tag{A-3}$$

left multiply $x(k)$ by e^{Ah}:

$$\begin{aligned}
e^{Ah}x(k) &= e^{A(k+1)h}x(0) + e^{A(k+1)h}\int_0^{kh} e^{-A\tau}B_1 + e^{-A\tau}B_2 u(\tau)d\tau \\
e^{A(k+1)h}x(0) &= e^{Ah}x(k) - e^{A(k+1)h}\int_0^{kh} e^{-A\tau}B_1 + e^{-A\tau}B_2 u(\tau)d\tau
\end{aligned} \tag{A-4}$$

now plug this result into Equation A-3

$$x(k+1) = e^{Ah}x(k) - e^{A(k+1)h}\int_0^{kh} e^{-A\tau}B_1 + e^{-A\tau}B_2 u(\tau)d\tau + e^{A(k+1)h}\int_0^{(k+1)kh} e^{-A\tau}B_1 + e^{-A\tau}B_2 u(\tau)d\tau$$

this simplifies to

$$x(k+1) = e^{Ah}x(k) - e^{A(k+1)h}\left(\int_0^{kh} e^{-A\tau}B_1 + e^{-A\tau}B_2 u(\tau)d\tau + \int_0^{(k+1)kh} e^{-A\tau}B_1 + e^{-A\tau}B_2 u(\tau)d\tau\right)$$

$$x(k+1) = e^{Ah}x(k) - e^{A(k+1)h}\left(\int_{kh}^{(k+1)h} e^{-A\tau}B_1 + e^{-A\tau}B_2 u(\tau)d\tau\right)$$

split the integral

$$x(k+1) = e^{Ah}x(k) - e^{A(k+1)h}\left(\int_{kh}^{(k+1)h} e^{-A\tau}B_1 d\tau + \int_{kh}^{(k+1)h} e^{-A\tau}B_2 u(\tau)d\tau\right)$$

bring the exponent in, and bring the constant terms outside of the integrals

$$x(k+1) = e^{Ah}x(k) - \left(\int_{kh}^{(k+1)h} e^{A((k+1)-\tau)}d\tau B_1 + \int_{kh}^{(k+1)h} e^{A((k+1)-\tau)}d\tau B_2 u(k)\right)$$

Introduce an auxiliary variable $v = (k+1)h - \tau$. This simplifies the integrals to

$$\begin{aligned} x(k+1) &= e^{Ah}x(k) - \left(\int_{h}^{0} e^{Av}dv B_1 + \int_{h}^{0} e^{Av}dv B_2 u(k)\right) \\ x(k+1) &= e^{Ah}x(k) + \int_{0}^{h} e^{Av}dv B_1 + \int_{0}^{h} e^{Av}dv B_2 u(k) \end{aligned} \tag{A-5}$$

Finally solving the integrals gives us the result:

$$x(k+1) = e^{Ah}x(k) + A^{-1}\left(e^{Ah} - I\right)B_1 + A^{-1}\left(e^{Ah} - I\right)B_2 u(k) \tag{A-6}$$

A-2 Parameters of the physical systems

This section contains a description and value of all the parameters used in the description of the physical systems.

Symbol	Value	Unit	Description
g	9.81	$\frac{m}{s^2}$	Gravitational constant
m	0.75	kg	Mass of the pendulum rod
L	1.0	m	Length of the pendulum rod
θ	-	rad	Angle of the pendulum rod
$\dot{\theta}$	-	rad/s	Angular velocity of the pendulum rod
u	-	N	Torque acting on the pendulum rod

Table A-1: Parameters of the inverted pendulum system

Symbol	Value	Unit	Description
g	9.81	$\frac{m}{s^2}$	Gravitational constant
m_1	0.75	kg	Total mass of pendulum rod 1
m_2	0.75	kg	Total mass of pendulum rod 2
L_1	1.0	m	Length of pendulum rod 1
L_2	1.0	m	Length of pendulum rod 2
c_1	0.75	m	position of center of gravity of rod 1
c_2	0.75	m	position of center of gravity of rod 1
θ_1	-	rad	Angle pendulum rod 1
θ_2	-	rad	Angle pendulum rod 2
$\dot{\theta}_1$	-	rad/s	Angular velocity of pendulum rod 1
$\dot{\theta}_2$	-	rad/s	Angular velocity of pendulum rod 2
u	-	N	Torque acting on pendulum rod 1

Table A-2: Parameters of the double pendulum system

Appendix B

Algorithms and code

B-1 Max-plus and min-plus layers

This code defines two custom layers, MaxPlus and MinPlus, using TensorFlow's Keras framework. These layers implement specialized multiplication operations: max-plus for MaxPlus and min-plus for MinPlus. Both layers take input data, weights, and optional biases to perform their respective operations

```
import tensorflow as tf
from tensorflow import keras
from keras.layers import Layer
class MaxPlus(Layer):
    def __init__(self, units=32, bias=True, input_dim=None, **kwargs):
        self.units = units
        self.bias = bias
        self.input_dim = input_dim
        if self.input_dim:
            kwargs["input_shape"] = (self.input_dim,)
        super().__init__(**kwargs)

    def build(self, input_shape):
        """Build the layer"""
        input_dim = input_shape[1]
        self.input_dim = input_dim
        self.b = self.add_weight(
            shape=(input_dim, self.units),
            initializer="random_normal",
            name='b',
            trainable=True,
        )
```

```
        if self.bias:
            self.extra_bias = self.add_weight(
                shape=(self.units,), initializer="zero", trainable=True
            )
        self.built = True

    def compute_output_shape(self, input_shape):
        """Compute the output shape of the layer"""
        return (input_shape[0], self.units)

    def call(self, inputs):
        """Execute the maxplus multiplication"""
        # repeat the input vector to match the number of units
        output = tf.repeat(
            tf.reshape(inputs, [-1, self.input_dim, 1]), repeats=self.units, axis=2
        )
        # add the weights and take the max
        output += self.b
        output = tf.reduce_max(output, axis=1)

        # add the bias if needed
        if self.bias:
            output += self.extra_bias
        return output

    def get_config(self):
        """" Get the config of the layer """
        config = super().get_config().copy()
        config.update(
            {
                "units": self.units,
                "bias": self.bias,
                "input_dim": self.input_dim,
            }
        )
        return config


class MinPlus(MaxPlus):
    def call(self, inputs):
        """Execute the maxplus multiplication"""
        output = tf.concat(
            [tf.reshape(inputs, [-1, self.input_dim, 1])] * self.units, 2
        )
        output += self.b
        output = tf.reduce_min(output, axis=1)
        if self.bias:
```

```
            output += self.extra_bias
        return output
```

B-2 Linearized MPC controller

The MPC class provides a method simple_mpc_step_u to perform one step of MPC optimization, aiming to find an optimal control action given a current state x. It relies on the perform_mpc_step method which is implemented by it's subclasses.

The LinearizedMPC class inherits from MPC and specializes in linearized MPC calculations. It initializes matrices required for quadratic programming optimization, constructs constraints and cost matrices, and provides a perform_mpc_step method to solve the optimization problem. The optimization aims to minimize a quadratic cost function, subject to linear inequality constraints. If the optimization succeeds, the method returns the optimal control action.

```
class MPC:
    def __init__(self, constants, horizon, control_horizon=None):
        self.constants = constants
        self.horizon = horizon
        self.control_horizon = control_horizon

    def simple_mpc_step_u(self, x, **kwargs):
        """Perform 1 mpc step with x = x0. Return optimal control"""
        u_opt, *args, success = self.perform_mpc_step(x, **kwargs)
        if success:
            if u_opt is not None:
                return u_opt[0][0]
            else:
                return float('NaN')
        else:
            return float('NaN')



class LinearizedMPC(MPC):
    def __init__(self, constants, horizon, control_horizon=None):
        super().__init__(constants, horizon, control_horizon=control_horizon)
        self.mpc_type = 'linearized'
        self.init_matrices()

    def init_matrices(self):
        constants = self.constants
        A, B, horizon = constants.SYS_DISCR.A, constants.SYS_DISCR.B, self.horizon
        Q, R, P = constants.MAT_Q, constants.MAT_R, constants.MAT_P
```

```
30          umax, umin, xmax, xmin = constants.UMAX, constants.UMIN, constants.XMAX,
            ↪  constants.XMIN
31          xterminal_min, xterminal_max = constants.x_terminal_min, constants.x_terminal_max
32          horizon = self.horizon
33          nx, nu = B.shape
34          # prepare matrices for quadratic programming problem
35          # state constraints
36          Abar = np.zeros((nx * horizon, nx))
37          Bbar = np.zeros((nx * horizon, nu * horizon))
38
39          for k in range(horizon):
40              Abar[nx * k:nx * k + nx, :] = np.linalg.matrix_power(A, k + 1)
41              for i in range(k + 1, 0, -1):
42                  Bbar[nx * k:nx * k + nx, i - 1:i] = np.linalg.matrix_power(A, k + 1 - i) @ B
43          if xmax is None:
44              xmaxbar = float('inf') * np.ones((horizon * nx, 1))
45          else:
46              xmaxbar = np.vstack([xmax] * horizon)
47          if xmin is None:
48              xminbar = float('-inf') * np.ones((horizon * nx, 1))
49          else:
50              xminbar = np.vstack([xmin] * horizon)
51          if umax is None:
52              umaxbar = float('inf') * np.ones((horizon * nu, 1))
53          else:
54              umaxbar = np.vstack([umax] * horizon)
55          if umin is None:
56              uminbar = float('-inf') * np.ones((horizon * nu, 1))
57          else:
58              uminbar = np.vstack([umin] * horizon)
59          if xterminal_max is not None:
60              xmaxbar[(horizon - 1) * nx:] = xterminal_max
61          if xterminal_min is not None:
62              xminbar[(horizon - 1) * nx:] = xterminal_min
63
64          self.constr = np.vstack([-Bbar, Bbar, np.eye(Bbar.shape[1]),
            ↪  -np.eye(Bbar.shape[1])])
65          self.umaxbar = umaxbar
66          self.uminbar = uminbar
67          self.xmaxbar = xmaxbar
68          self.xminbar = xminbar
69          self.Abar = Abar
70          self.Bbar = Bbar
71          Q_list = [Q] * (horizon - 1)
72          Q_list.append(P)
73          self.Qbar = scipy.linalg.block_diag(*Q_list)
74          self.Rbar = scipy.linalg.block_diag(*([R] * horizon))
```

```
        self.cross_matrix = Abar.T @ self.Qbar @ self.Bbar
        self.ucost = Bbar.T @ self.Qbar @ Bbar + self.Rbar

    def perform_mpc_step(self, x_0):
        """Solve a strictly convex quadratic program
        Minimize     1/2 x^T G x - a^T x
        Subject to   C.T x >= b
        """
        # Setup u_variables
        cross_cost_vec = x_0.T @ self.cross_matrix
        cross_cost_vec = -2 * cross_cost_vec.flatten()
        Ax0 = self.Abar @ x_0.reshape(-1, 1)
        constr_d = np.vstack([Ax0 - self.xmaxbar, self.xminbar - Ax0, self.uminbar,
        ↪  -self.umaxbar]).flatten()
        try:
            u_sol = quadprog.solve_qp(self.ucost, cross_cost_vec, self.constr.T, constr_d)
            return u_sol, 'pass', True
        except:
            return np.array([[0]]), 'fail', False
```

B-3 PWA MPC controller

This code defines a class PiecewiseAffineMPC for MPC tailored to systems represented by piecewise-affine models. The class includes methods to perform MPC steps, create piecewise affine approximations of functions, and generate constraint matrices for the MPC optimization problem.

```
class PiecewiseAffineMPC(MPC):
    def __init__(self, constants, horizon, control_horizon=None):
        super().__init__(constants, horizon, control_horizon=control_horizon)
        self.mpc_type = "pwa"
        self.affine_functions, self.breakpoints = self.create_pwa_approx()
        (
            self.E1,
            self.E2,
            self.E3,
            self.E4,
            self.G5,
        ) = self.generate_mld_constraint_matrices()
        self.B3 = self.create_mld_matrices()

    def perform_mpc_step(self, x_0, verbose=False, solver="GUROBI", reoptimize=True,
    ↪  Cuts=3,speedup=True, **kwargs):
        """Perform mpc step"""
```

```
        # Define variables
        if speedup:
            if x_0[1] <(-5*np.pi/2)*x_0[0]-np.pi:
                ans = 9.99
                return (
                    np.array([[ans]]),
                    np.array([[0]]),
                    np.array([[0]]),
                    np.array([[0]]),
                    True,
                )
            if x_0[1] > (-5*np.pi/1.9)*x_0[0]+np.pi:
                ans = -9.99
                return (
                    np.array([[ans]]),
                    np.array([[0]]),
                    np.array([[0]]),
                    np.array([[0]]),
                    True,
                )

        constants = self.constants
        Q, R, P = constants.MAT_Q, constants.MAT_R, constants.MAT_P
        umax, umin, xmax, xmin = (
            constants.UMAX,
            constants.UMIN,
            constants.XMAX,
            constants.XMIN,
        )
        E1, E2, E3, E4, G5 = self.E1, self.E2, self.E3, self.E4, self.G5
        B3 = self.B3
        nx = 2
        nu = 1
        nr = self.constants.N_REGIONS
        u = cvxpy.Variable((nu, self.horizon))
        delta = cvxpy.Variable((nr, self.horizon + 1), boolean=True)
        z = cvxpy.Variable((nr * nx, self.horizon + 1))

        epsilon_u = cvxpy.Variable((nu, self.horizon))
        epsilon_x = cvxpy.Variable((nx, self.horizon + 1))
        ones_m = np.ones((1, nu))
        ones_n = np.ones((1, nx))

        # initialize cost and constraints
        cost = 0.0
        constr = []
        init_delta1 = [1 if x_0[0] >= bp else 0 for bp in self.breakpoints[:-1]]
```

```
64          init_delta2 = [1 if x_0[0] <= bp else 0 for bp in self.breakpoints[1:]]
65          init_delta3 = [True if (init_delta1[i] == 1 and init_delta2[i] == 1) else False for
            ↪  i in
66                         range(len(init_delta1))]
67          constr += [delta[:, 0] == init_delta3[:]]
68          constr += [z[:, 0] == (x_0.reshape(2, 1) * (init_delta3 * np.ones((2,
            ↪  1)))).T.flatten()]
69          constr += [epsilon_x[:, 0] == np.abs(x_0)]
70          N = self.horizon
71
72          for k in range(1, N + 1):
73              constr += [-ones_m @ epsilon_u[:, k - 1] <= R @ u[:, k - 1]]
74              constr += [-ones_m @ epsilon_u[:, k - 1] <= -R @ u[:, k - 1]]
75              constr += [-ones_n @ epsilon_x[:, k] <= Q @ B3 @ z[:, k - 1]]
76              constr += [-ones_n @ epsilon_x[:, k] <= -Q @ B3 @ z[:, k - 1]]
77              constr += [u[:, k - 1] <= self.constants.UMAX]
78              constr += [-u[:, k - 1] <= -self.constants.UMIN]
79
80              constr += [
81                  E1 @ B3 @ z[:, k - 1] + E2 @ u[:, k - 1] + E3 @ delta[:, k] + E4 @ z[:, k]
82                  <= G5[:, 0]
83              ]
84
85          # constr += [x[:, N] == B3 @ z[:, N - 1]]
86          constr += [-ones_n @ epsilon_x[:, N] <= P @ B3 @ z[:, N - 1]]
87          constr += [-ones_n @ epsilon_x[:, N] <= -P @ B3 @ z[:, N - 1]]
88          constr += [B3 @ z[:, N - 1] <= np.array([0.1, 0.1])]
89          constr += [-B3 @ z[:, N - 1] <= np.array([0.1, 0.1])]
90          for i in range(N + 1):
91              cost += cvxpy.pnorm(epsilon_x[:, i], 1)
92              if i == N:
93                  break
94              cost += cvxpy.pnorm(epsilon_u[:, i],1)
95
96          try:
97              prob = cvxpy.Problem(cvxpy.Minimize(cost), constr)
98              prob.solve(
99                  verbose=verbose, solver=solver, reoptimize=reoptimize, time_limit=10,
                    ↪  ConcurrentMIP=2, **kwargs,
100             )
101         except Exception as e:
102             return (
103                 np.array([0]),
104                 np.array([[0]]),
105                 np.array([[0]]),
106                 np.array([[0]]),
107                 False,
```

```
            )

        if prob.status not in ["infeasible", "unbounded"]:
            return u.value, 3, delta.value, z.value, True
        else:
            return (
                np.array([0]),
                np.array([[0]]),
                np.array([[0]]),
                np.array([[0]]),
                False,
            )

    def create_pwa_approx(self, func=np.sin):
        """create pwa approximate of function, returns affine functions ai and bi in
        ↪  tuple"""
        # Equally divided breakpoints
        n_regions = self.constants.N_REGIONS
        breakpoints = np.linspace(-0.5 * np.pi, 0.5 * np.pi, n_regions + 1)
        affine_functions = []
        for i, bp in enumerate(breakpoints):
            if i == len(breakpoints) - 1:
                break
            y1, y2 = func(bp), func(breakpoints[i + 1])
            x1, x2 = bp, breakpoints[i + 1]
            ai = (y2 - y1) / (x2 - x1)
            bi = y2 - ai * x2
            affine_functions.append((ai, bi))
        return affine_functions, breakpoints

    def create_mld_matrices(self):
        """create matrices for the mld problem"""
        # create matrices
        B3 = np.zeros((2, 2 * self.constants.N_REGIONS))
        B3[0, ::2] = 1
        B3[1, 1::2] = 1
        return B3

    def generate_mld_constraint_matrices(self):
        """create constraint matrices for the mld problem"""
        # create constraint matrices
        constants = self.constants
        T_s, mass, length = constants.T_s, constants.M, constants.L
        gravitational_constant = constants.G
        nx = 2
        nu = 1
        nr = self.constants.N_REGIONS
```

```
154         E1 = np.zeros((0, nx))  # x(t) constraints
155         E2 = np.zeros((0, nu))  # u(t) constraints
156         E3 = np.zeros((0, nr))  # delta(t) constraints
157         E4 = np.zeros((0, nr * nx))  # z(t) constraints
158         G5 = np.zeros((0, 1))  # constant constraints
159         constants = self.constants
160         umax, umin, xmax, xmin = (
161             constants.UMAX,
162             constants.UMIN,
163             constants.XMAX,
164             constants.XMIN,
165         )
166 
167         if xmin is None:
168             xmin = -99999999
169         if xmax is None:
170             xmax = 99999999
171         theta_is = self.breakpoints
172         M_star = 100 * np.ones((nx, 1))
173         m_star = -100 * np.ones((nx, 1))
174 
175         for i in range(len(theta_is) - 1):
176             n_constraints = nx * nr + 2 * nu + 2 * nx
177             E1_row = np.zeros((n_constraints, nx))
178             E2_row = np.zeros((n_constraints, nu))
179             E3_row = np.zeros((n_constraints, nr))
180             E4_row = np.zeros((n_constraints, nr * nx))
181             G5_row = np.zeros((n_constraints, 1))
182 
183             # constraints 1,2,.. nx
184             # Sx + Ru+ Mdelta <= M+Ti
185             E1_row[0:nx, :] = np.array([[1, 0], [-1, 0]])
186             E3_row[0:nx, i] = M_star.flatten()
187             G5_row[0:nx, :] = np.array([[theta_is[i + 1]], [-theta_is[i]]]) + M_star
188             #
189             # constraints nx+1, nx+2,.. 2nx
190             # -Mdelta +z <= 0
191             E3_row[nx:2 * nx, i] = -M_star.flatten()
192             E4_row[nx:2 * nx, i * nx:(i + 1) * nx] = np.eye(nx)  # np.ones((nx, nx))
193             #
194             # constraints 2nx+1, 2nx+2,.. 3nx
195             # mdelta -z <= 0
196             E3_row[2 * nx:3 * nx, i] = m_star.flatten()
197             E4_row[2 * nx:3 * nx, i * nx:(i + 1) * nx] = -np.eye(nx)  # -np.ones((nx, nx))
198 
199             # compute matrices for constraints
200             cont_Ai = np.array(
```

```
                [[0, 1], [(3 * gravitational_constant / 2 * length) *
                ↪  self.affine_functions[i][0], 0]]
            )
            discrete_Ai = scipy.linalg.expm(cont_Ai * T_s)
            preamble = scipy.linalg.inv(cont_Ai) @ (discrete_Ai - np.eye(2))
            Fi = preamble @ np.array([[0], [self.affine_functions[i][1]]])
            Bi = preamble @ np.array([[0], [3 / (mass * length * length)]])

            # constraints 3nx+1, 3nx+2,.. 4nx
            # -Ax -Bu - mdelta +z <=-m+F
            E1_row[3 * nx:4 * nx, :] = -discrete_Ai
            E2_row[3 * nx:4 * nx, :] = -Bi
            E3_row[3 * nx:4 * nx, i] = -m_star.flatten()
            E4_row[3 * nx:4 * nx, i * nx:(i + 1) * nx] = np.eye(nx)  # np.ones((nx, nx))
            G5_row[3 * nx:4 * nx, :] = (-m_star + Fi)

            # constraints 4nx+1, 4nx+2,.. 5nx
            # Ax +Bu + Mdelta -z <= M-F
            E1_row[4 * nx:5 * nx, :] = discrete_Ai
            E2_row[4 * nx:5 * nx, :] = Bi
            E3_row[4 * nx:5 * nx, i] = M_star.flatten()
            E4_row[4 * nx:5 * nx, i * nx:(i + 1) * nx] = -np.eye(nx)  # -np.ones((nx, nx))
            G5_row[4 * nx:5 * nx, :] = M_star - Fi

            # constraints 5nx+1, 5nx+2,.. 6nx
            # x <= xmax
            E1_row[5 * nx:6 * nx, :] = np.eye(nx)
            G5_row[5 * nx:6 * nx, :] = xmax

            # constraints 6nx+1, 6nx+2,.. 7nx
            # -x <= -xmin
            E1_row[6 * nx:7 * nx, :] = -np.eye(nx)
            G5_row[6 * nx:7 * nx, :] = -xmin

            # stack the rows to the matrices
            E1 = np.vstack((E1, E1_row))
            E2 = np.vstack((E2, E2_row))
            E3 = np.vstack((E3, E3_row))
            E4 = np.vstack((E4, E4_row))
            G5 = np.vstack((G5, G5_row))
        # add one more constraint for all regions
        E1 = np.vstack((E1, np.zeros((1, nx))))
        E2 = np.vstack((E2, np.zeros((1, nu))))
        E3 = np.vstack((E3, np.ones((1, nr))))
        E4 = np.vstack((E4, np.zeros((1, nr * nx))))
        G5 = np.vstack((G5, np.ones((1, 1))))

```

```
247         E1 = np.vstack((E1, np.zeros((1, nx))))
248         E2 = np.vstack((E2, np.zeros((1, nu))))
249         E3 = np.vstack((E3, -np.ones((1, nr))))
250         E4 = np.vstack((E4, np.zeros((1, nr * nx))))
251         G5 = np.vstack((G5, -np.ones((1, 1))))
252 
253         return E1, E2, E3, E4, G5
```

B-4 Scoring metric

This code implements the custom scoring method that uses a specified norm to compute a cost on the states and inputs.

```
1 
2 def score_results(result, P, Q, R, filter_unstable=False):
3     state_cost_list = []
4     input_cost_list = []
5     counter = 0
6     total = len(result)
7     for res in result:
8         y = res[0]
9         u = res[1].reshape(-1, 1)
10         if filter_unstable:
11             print(y.T[0])
12             if np.any((y.T[0] < -0.9) | (y.T[0] > 0.9)) or np.any((y.T[1] < -1.3) | (y.T[1]
               ↪  > 1.3)):
13                 counter += 1
14                 continue
15         state_cost_list.append(0)
16         input_cost_list.append(0)
17         for i in range(len(y) - 1):
18             state_cost_list[-1] += y[i].T @ Q @ y[i]
19             input_cost_list[-1] += u[i].T @ R @ u[i]
20         state_cost_list[-1] += y[-1].T @ P @ y[-1]
21         input_cost_list[-1] += u[-1].T @ R @ u[-1]
22 
23     state_cost_list = np.array(state_cost_list)
24     state_cost_mean = np.mean(state_cost_list)
25     input_cost_list = np.array(input_cost_list)
26     input_cost_mean = np.mean(input_cost_list)
27     mean_cost_both = state_cost_mean + input_cost_mean
28     return state_cost_mean, input_cost_mean, mean_cost_both, counter, total,
       ↪  state_cost_list, input_cost_list
29 
```

```
30
31  def score_results_pnorm(result, P, Q, R, pnorm=1, filter_unstable=False):
32      state_cost_list = []
33      input_cost_list = []
34      counter = 0
35      total = len(result)
36      for res in result:
37          y = res[0]
38          u = res[1].reshape(-1, 1)
39          if filter_unstable:
40              if np.any((y.T[0] < -0.7) | (y.T[0] > 0.7)) or np.any((y.T[1] < -1) | (y.T[1] >
                ↪  1)):
41                  counter += 1
42                  continue
43          state_cost_list.append(0)
44          input_cost_list.append(0)
45          for i in range(len(y) - 1):
46              #
47              state_cost_list[-1] += np.linalg.norm(Q @ y[i], pnorm)
48              input_cost_list[-1] += np.linalg.norm(R @ u[i], pnorm)
49          state_cost_list[-1] += np.linalg.norm(P @ y[-1], pnorm)
50          input_cost_list[-1] += np.linalg.norm(u[-1], pnorm)
51
52      state_cost_list = np.array(state_cost_list)
53      state_cost_mean = np.mean(state_cost_list)
54      input_cost_list = np.array(input_cost_list)
55      input_cost_mean = np.mean(input_cost_list)
56      mean_cost_both = state_cost_mean + input_cost_mean
57      return state_cost_mean, input_cost_mean, mean_cost_both, counter, total,
        ↪  state_cost_list, input_cost_list
```

Bibliography

[1] C. E. Garcia and M. Morari, "Internal model control. a unifying review and some new results," *Industrial & Engineering Chemistry Process Design and Development*, vol. 21, no. 2, pp. 308–323, 1982. DOI: 10.1021/i200017a016. eprint: https://doi.org/10.1021/i200017a016. [Online]. Available: https://doi.org/10.1021/i200017a016.

[2] J. Richalet, "Industrial applications of model based predictive control," *Automatica*, vol. 29, no. 5, pp. 1251–1274, 1993, ISSN: 0005-1098. DOI: https://doi.org/10.1016/0005-1098(93)90049-Y. [Online]. Available: https://www.sciencedirect.com/science/article/pii/000510989390049Y.

[3] A. Alessio and A. Bemporad, "A survey on explicit model predictive control," in *Nonlinear Model Predictive Control: Towards New Challenging Applications*, L. Magni, D. M. Raimondo, and F. Allgöwer, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 345–369, ISBN: 978-3-642-01094-1. DOI: 10.1007/978-3-642-01094-1_29. [Online]. Available: https://doi.org/10.1007/978-3-642-01094-1_29.

[4] W. Heemels, B. De Schutter, and A. Bemporad, "On the equivalence of classes of hybrid dynamical models," in *Proceedings of the 40th IEEE Conference on Decision and Control (Cat. No. 01CH37228)*, IEEE, vol. 1, 2001, pp. 364–369.

[5] Y. Luo and S. Fan, *Min-max-plus neural networks*, 2021. arXiv: 2102.06358 [cs.NE].

[6] B. M. Åkesson and H. T. Toivonen, "A neural network model predictive controller," *Journal of Process Control*, vol. 16, no. 9, pp. 937–946, 2006, ISSN: 0959-1524. DOI: https://doi.org/10.1016/j.jprocont.2006.06.001. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0959152406000618.

[7] B. B. Schwedersky and R. C. Flesch, "Nonlinear model predictive control algorithm with iterative nonlinear prediction and linearization for long short-term memory network models," *Engineering Applications of Artificial Intelligence*, vol. 115, p. 105 247, 2022, ISSN: 0952-1976. DOI: https://doi.org/10.1016/j.engappai.2022.105247. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0952197622003177.

[8] E. Maddalena, C. da S. Moraes, G. Waltrich, and C. Jones, "A neural network architecture to learn explicit mpc controllers from data," *IFAC-PapersOnLine*, vol. 53, no. 2, pp. 11 362–11 367, 2020, 21st IFAC World Congress, ISSN: 2405-8963. DOI: https://doi.org/10.1016/j.ifacol.2020.12.546. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2405896320308442.

[9] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[10] Y. Pan and J. Wang, "Model predictive control of unknown nonlinear dynamical systems based on recurrent neural networks," *IEEE Transactions on Industrial Electronics*, vol. 59, no. 8, pp. 3089–3101, 2011.

[11] T. van den Boom, G. Abhimanyu, and B. De Schutter, *Internal Report*. Sep. 2022.

[12] E. Sontag, "Nonlinear regulation: The piecewise linear approach," *IEEE Transactions On Automatic Control*, vol. 26, no. 2, pp. 346–358, 1981.

[13] A. Bemporad and M. Morari, "Control of systems integrating logic, dynamics, and constraints," *Automatica*, vol. 35, no. 3, pp. 407–427, 1999.

[14] A. van der Schaft and J. Schumacher, "Hybrid systems modeling and complementarity problems," English, in *Proceedings of the European Control Conference, Brussels* (CDRom 868), CDRom 868. CDRom 868, 1997.

[15] W. Heemels, J. M. Schumacher, and S. Weiland, "Linear complementarity systems," *SIAM Journal On Applied Mathematics*, vol. 60, no. 4, pp. 1234–1269, 2000.

[16] B. De Schutter, "Optimal control of a class of linear hybrid systems with saturation," *SIAM Journal on Control and Optimization*, vol. 39, no. 3, pp. 835–851, 2000.

[17] B. De Schutter and T. J. van den Boom, "Mpc for continuous piecewise-affine systems," *Systems & Control Letters*, vol. 52, no. 3-4, pp. 179–192, 2004.

[18] M. Wild, "Idempotent and co-idempotent stack filters and min–max operators," *Theoretical Computer Science*, vol. 299, no. 1-3, pp. 603–631, 2003.

[19] A. Kripfganz and R. Schulze, "Piecewise affine functions as a difference of two convex functions," *Optimization*, vol. 18, no. 1, pp. 23–29, 1987. DOI: 10.1080/02331938708843210.

[20] J. Xu, T. J. van den Boom, B. De Schutter, and S. Wang, "Irredundant lattice representations of continuous piecewise affine functions," *Automatica*, vol. 70, pp. 109–120, 2016.

[21] R. A. Cuninghame-Green and P. Meijer, "An algebra for piecewise-linear minimax problems," *Discrete Applied Mathematics*, vol. 2, no. 4, pp. 267–294, 1980.

[22] F. Baccelli, G. Cohen, G. J. Olsder, and J.-P. Quadrat, "Synchronization and linearity: An algebra for discrete event systems," 1992.

[23] B. Heidergott, G. J. Olsder, J. Van Der Woude, and J. van der Woude, *Max Plus at work: modeling and analysis of synchronized systems: a course on Max-Plus algebra and its applications*. Princeton University Press, 2006, vol. 13.

[24] B. De Schutter and T. van den Boom, "Max-plus algebra and max-plus linear discrete event systems: An introduction," in *Proceedings of the 9th International Workshop on Discrete Event Systems (WODES'08)*, Göteborg, Sweden, May 2008, pp. 36–42.

[25] R. E. Kálmán, "Contributions to the theory of optimal control," 1960.

[26] D. Goldfarb and A. Idnani, "A numerically stable dual method for solving strictly convex quadratic programs," *Mathematical Programming*, vol. 27, no. 1, pp. 1–33, 1983.

[27] J. Rawlings, D. Mayne, and M. Diehl, *Model Predictive Control: Theory, Computation, and Design*, 2nd ed. Nob Hill Publishing, 2017, ISBN: 9780975937730.

[28] F. Borrelli, A. Bemporad, and M. Morari, *Predictive control for linear and hybrid systems*. Cambridge University Press, 2017.

[29] A. Bemporad, M. Morari, V. Dua, and E. N. Pistikopoulos, "The explicit linear quadratic regulator for constrained systems," *Automatica*, vol. 38, no. 1, pp. 3–20, 2002.

[30] S. Sharma, S. Sharma, and A. Athaiya, "Activation functions in neural networks," *Towards Data Acience*, vol. 6, no. 12, pp. 310–316, 2017.

[31] D.-A. Clevert, T. Unterthiner, and S. Hochreiter, *Fast and accurate deep network learning by exponential linear units (elus)*, 2016. arXiv: 1511.07289 [cs.LG].

[32] D. Hendrycks and K. Gimpel, *Gaussian error linear units (gelus)*, 2023. arXiv: 1606.08415 [cs.LG].

[33] G. Klambauer, T. Unterthiner, A. Mayr, and S. Hochreiter, *Self-normalizing neural networks*, 2017. arXiv: 1706.02515 [cs.LG].

[34] M. Courbariaux, Y. Bengio, and J.-P. David, *Binaryconnect: Training deep neural networks with binary weights during propagations*, 2016. arXiv: 1511.00363 [cs.LG].

[35] M. Zhou, *Softplus regressions and convex polytopes*, 2016. arXiv: 1608.06383 [stat.ML].

[36] J. Van der Woude, "A characterization of the eigenvalue of a general (min, max,+)-system," *Discrete Event Dynamic Systems*, vol. 11, pp. 203–210, 2001.

[37] T. Serra, C. Tjandraatmadja, and S. Ramalingam, "Bounding and counting linear regions of deep neural networks," in *Proceedings of the 35th International Conference on Machine Learning*, J. Dy and A. Krause, Eds., ser. Proceedings of Machine Learning Research, vol. 80, PMLR, Jul. 2018, pp. 4558–4566. [Online]. Available: https://proceedings.mlr.press/v80/serra18b.html.

[38] T. Zaslavsky, *Facing up to arrangements: Face-count formulas for partitions of space by hyperplanes: Face-count formulas for partitions of space by hyperplanes*. American Mathematical Soc., 1975, vol. 154.

[39] G. F. Montúfar, R. Pascanu, K. Cho, and Y. Bengio, "On the number of linear regions of deep neural networks," *Advances In Neural Information Processing Systems*, vol. 27, 2014.

[40] M. Raghu, B. Poole, J. Kleinberg, S. Ganguli, and J. Sohl-Dickstein, "On the expressive power of deep neural networks," in *International Conference On Machine Learning*, PMLR, 2017, pp. 2847–2854.

[41] G. Montúfar, "Notes on the number of linear regions of deep neural networks," 2017.

[42] E. Danna, M. Fenelon, Z. Gu, and R. Wunderling, "Generating multiple solutions for mixed integer programming problems," in *Integer Programming and Combinatorial Optimization: 12th International IPCO Conference, Ithaca, NY, USA, June 25-27, 2007. Proceedings 12*, Springer, 2007, pp. 280–294.

[43] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, no. 6088, pp. 533–536, 1986.

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

[45] Y. Nesterov, "A method for unconstrained convex minimization problem with the rate of convergence o(1/k2)," in *Doklady an USSR*, vol. 269, 1983, pp. 543–547.

[46] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization.," *Journal of Machine Learning Research*, vol. 12, no. 7, 2011.

[47] D. P. Kingma and J. Ba, *Adam: A method for stochastic optimization*, 2014. DOI: 10.48550/ARXIV.1412.6980. [Online]. Available: https://arxiv.org/abs/1412.6980.

[48] K. Hornik, M. Stinchcombe, and H. White, "Multilayer feedforward networks are universal approximators," *Neural Networks*, vol. 2, no. 5, pp. 359–366, 1989.

[49] B. Hanin, "Universal function approximation by deep neural nets with bounded width and relu activations," *Mathematics*, vol. 7, no. 10, p. 992, 2019.

[50] B. Karg and S. Lucia, "Efficient representation and approximation of model predictive control laws via deep learning," *IEEE Transactions on Cybernetics*, vol. 50, no. 9, pp. 3866–3878, 2020.

[51] S. Piche, B. Sayyar-Rodsari, D. Johnson, and M. Gerules, "Nonlinear model predictive control using neural networks," *IEEE Control Systems Magazine*, vol. 20, no. 3, pp. 53–62, 2000. DOI: 10.1109/37.845038.

[52] K. Kiš and M. Klaučo, *Neural network based explicit mpc for chemical reactor control*, 2019. arXiv: 1912.04684 [cs.LG].

[53] R. May, H. Maier, and G. Dandy, "Data splitting for artificial neural networks using som-based stratified sampling," *Neural Networks*, vol. 23, no. 2, pp. 283–294, 2010, ISSN: 0893-6080. DOI: https://doi.org/10.1016/j.neunet.2009.11.009. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0893608009002949.

[54] L. Prechelt, "Early stopping - but when?" In *Neural Networks: Tricks of the Trade*, G. B. Orr and K.-R. Müller, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 55–69, ISBN: 978-3-540-49430-0. DOI: 10.1007/3-540-49430-8_3. [Online]. Available: https://doi.org/10.1007/3-540-49430-8_3.

[55] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang, *JAX: Composable transformations of Python+NumPy programs*, version 0.3.13, 2018. [Online]. Available: http://github.com/google/jax.

[56] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, *TensorFlow: Large-scale machine learning on heterogeneous systems*, Software available from tensorflow.org, 2015. [Online]. Available: https://www.tensorflow.org/.

[57] L. Keviczky, R. Bars, J. Hetthéssy, and C. Bányász, *Control Engineering*. Springer, 2019.

[58] H. Vallery and L. A. Schwab, *Advanced Dynamics*. Stichting Newton-Euler, 2018, ch. 13.

[59] Intel, *Intel 64 and ia-32 architectures optimization reference manual*, Order Number: 248966-046A, Jan. 2023.

[60] M. Wang, S. Rasoulinezhad, P. H. W. Leong, and H. K.-H. So, "NITI: Training integer neural networks using integer-only arithmetic," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 11, pp. 3249–3261, Nov. 2022. DOI: 10.1109/tpds.2022.3149787. [Online]. Available: https://doi.org/10.1109%2Ftpds.2022.3149787.

[61] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *The Journal of Machine Learning Research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[62] H. Dai, B. Landry, L. Yang, M. Pavone, and R. Tedrake, *Lyapunov-stable neural-network control*, 2021. arXiv: 2109.14152 [cs.RO].

[63] B. Karg and S. Lucia, "Stability and feasibility of neural network-based controllers via output range analysis," in *2020 59th IEEE Conference on Decision and Control (CDC)*, 2020, pp. 4947–4954. DOI: 10.1109/CDC42340.2020.9303895.

[64] J. Hendriks, C. Jidling, A. Wills, and T. Schön, *Linearly constrained neural networks*, 2021. arXiv: 2002.01600 [stat.ML].

[65] G. Schullerus, V. Krebs, B. De Schutter, and T. van den Boom, "Input signal design for identification of max-plus-linear systems," *Automatica*, vol. 42, no. 6, pp. 937–943, 2006, ISSN: 0005-1098. DOI: https://doi.org/10.1016/j.automatica.2006.01.025. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0005109806000938.