DEGREE PROJECT IN ELECTRICAL ENGINEERING,
SECOND CYCLE, 30 CREDITS
STOCKHOLM, SWEDEN 2020

Neural Network Approaches for Model Predictive Control

REBECKA WINQVIST

KTH ROYAL INSTITUTE OF TECHNOLOGY
ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Abstract

Model Predictive Control (MPC) is an optimization-based paradigm for feedback control. The MPC relies on a dynamical model to make predictions for the future values of the controlled variables of the system. It then solves a constrained optimization problem to calculate the optimal control action that minimizes the difference between the predicted values and the desired or set values. One of the main limitations of the traditional MPC lies in the high computational cost resulting from solving the associated optimization problem online. Various offline strategies have been proposed to overcome this, ranging from the explicit MPC (eMPC) to the recent learning-based neural network approaches. This thesis investigates a framework for the training and evaluation of a neural network for learning to implement the MPC. As a part of the framework, a new approach for efficient generation of training data is proposed. Four different neural network structures are studied; one of them is a black box network while the other three employ MPC specific information. The networks are evaluated in terms of two different performance metrics through experiments conducted on realistic two-dimensional and four-dimensional systems. The experiments reveal that while using MPC specific structure in the neural networks results in performance gains when the training data is limited, all the network structures perform similarly as extensive training data is used. They further show that a recurrent neural network structure trained on both the state and control trajectories of a family of MPCs is able to generalize to previously unseen MPC problems. The proposed methods in this thesis act as a first step towards developing a coherent framework for characterization of learning approaches in terms of both model validation and efficient training data generation.

Sammanfattning

Modell-prediktiv reglering (MPC) är en strategi inom återkopplad reglering med rötter i optimeringsteori. MPC:n använder sig av en dynamisk modell för att prediktera de framtida värdena på systemets styrvariabler. Den löser sedan ett optimeringsproblem för att beräkna en optimal styrsignal som minimerar skillnaden mellan referensvärdena och de predikterade värdena. Att lösa det associerade optimeringsproblemet online kan medföra höga beräkningskostnader, något som utgör en av de huvudsakliga begränsningarna med traditionell MPC. Olika offline-strategier har föreslagits för att kringgå detta, däribland explicit modell-prediktiv reglering (eMPC) samt senare inlärningsmetoder baserade på neuronnät. Den här masteruppsatsen undersöker ett ramverk för träning och utvärdering av olika neuronnätsstrukturer för MPC-inlärning. En ny metod för effektiv generering av träningsdata presenteras som en del av detta ramverk. Fyra olika nätstrukturer studeras; ett black box-nät samt tre nät som inkluderar MPC-specifik information. Näten evalueras i termer av två olika prestandamått genom experiment på realistiska två- och fyrdimensionella system. Experimenten visar att en MPC-specifik nätstruktur resulterar i ökad prestanda när mängden träningsdata är begränsad, men att de fyra näten presterar likvärdigt när mycket träningsdata finns att tillgå. De visar vidare att ett återkopplat neuronnät som tränas på både tillstånds- och styrsignalstrajektorier från en familj av MPC:er har förmågan att generalisera vid påträffandet av nya MPC-problem. De föreslagna metoderna i den här uppsatsen utgör ett första steg mot utvecklandet av ett enhetligt ramverk för karaktärisering av inlärningsmetoder i termer av både modellvalidering och effektiv datagenerering.

Acknowledgements

I would like to express my gratitude to my supervisors, Prof. Bo Wahlberg and Dr. Arun Venkitaraman, for their invaluable input and support throughout this project. I am especially thankful for all the effort they have put in to make this an experience out of the ordinary.

Contents

Abbrevations

BBNN	Black Box Neural Network
CLQR	Constrained Linear Quadratic Regulation
DPP	Disciplined Parametrized Programming
eMPC	Explicit Model Predictive Control
FFNN	Feed-Forward Neural Network
HAR	Hit-And-Run
LQ	Linear-Quadratic
LQR	Linear-Quadratic Regulator
LQR-PNN	LQR Projection Neural Network
LSTM	Long Short-Term Memory
LSTM-RNN	LSTM Recurrent Neural Network
MPC	Model Predictive Control
MSE	Mean Squared Error
NMSE	Normalized Mean Squared Error
PNN	Projection Neural Network
PWA	Piece-Wise Affine
ReLU	Rectified Linear Unit
RNN	Recurrent Neural Network

Symbols

$\mathbb{A} \subset \mathbb{B}$	Set $\mathbb{A}$ is a proper subset of $\mathbb{B}$
$\mathbb{A} \subseteq \mathbb{B}$	Set $\mathbb{A}$ is a subset of $\mathbb{B}$
$\mathbf{A} \succ 0$	Matrix $\mathbf{A}$ is positive definite
$\mathbf{A} \succeq 0$	Matrix $\mathbf{A}$ is positive semi-definite
$\mathcal{C}_\infty$	Maximal control invariant set
$\nabla f(\cdot)$	Gradient of $f(\cdot)$

Chapter 1

Introduction

This thesis project is concerned with the implementation and evaluation of neural network-based Model Predictive Control (MPC) controllers. While much attention has been given to this research topic in the past two decades, a consistent analysis methodology for learning-based MPC approaches is still lacking in literature. One recognizes that a systematic treatment of said approaches is needed for such regulator structures to gain success in industry and, in particular, for applications concerning safety-critical systems. This project is a step in this direction, with parts of this work having resulted in the contribution [44].

The remainder of this report will cover the topics involved in the process of learning the MPC control law, which is visualized in Figure 1.0.1. It will deal with the construction of different neural network structures as well as the implementation of an efficient method for sampling the state space for generating training data. It also covers performance evaluation of the implemented network controllers trough experiments conducted on realistic systems.

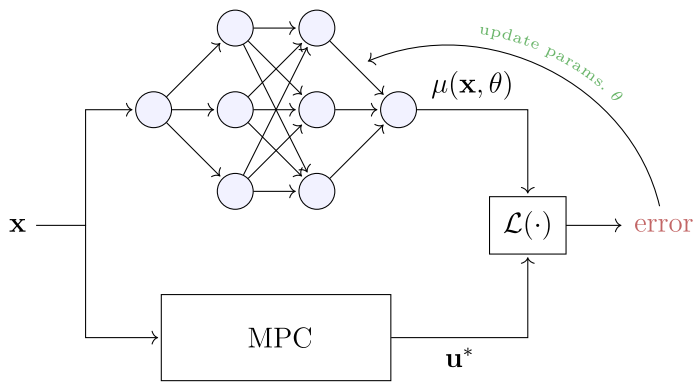


Figure 1.0.1: Visualization of learning the MPC, where $\mathbf{x}$ is a sampled state, $\mathbf{u}^*$ the corresponding optimal control action, $\mu(\mathbf{x}, \theta)$ the control action generated by the network, and $\mathcal{L}(\cdot)$ the loss function comparing the two control actions.

1.1 Background and Motivation

One of the challenges in the design of feedback control of safety critical systems is to ensure that the closed loop system always satisfies the given specifications. This is well understood when using classical control concepts such as PID and Linear-Quadratic Regulator (LQR) control. It is less understood when it comes to MPC. The lack of rigorous methods for verification of MPC often becomes a deterring factor against its widespread use in the vehicle industry despite its great potential. This in turn has prompted extensive research efforts in certified real-time optimization for MPC over the recent years, see [12] for an overview.

The MPC is typically formulated as a constrained, finite-horizon optimization program with a quadratic cost subject to linear constraints, which handle both the model of the system under study as well as the state and input constraints. The approach for synthesizing the control law entails solving the optimization problem at each time instant repeatedly over a receding time horizon. This procedure may quickly become computationally intractable as the number of states and constraints grows, which often limits the applicability of the MPC.

A strategy for circumventing this issue is through offline pre-computation of the control law. This transforms the problem into that of specifying a mapping or lookup table in the form of a Piece-Wise Affine (PWA) function defined over partitions of the state space. This approach is known as the explicit MPC [3, 17]. The explicit MPC has further inspired the viewing of the MPC problem as a general learning problem. As a result, a number of works involving the use of learning approaches, primarily in the form of artificial neural networks have been proposed recently [10, 11, 31].

While the idea of using neural networks based MPC is by no means novel, there are still many challenges with such learning based approaches, both in terms of the modelling, training and evaluation. In particular, it is not obvious how to efficiently generate training data for learning the control law as a mapping. Earlier work employ either grid-based methods or random sampling, neither which scales well to high dimensions.

Furthermore, it remains unclear as to how different network structures can be evaluated consistently. There is a need for a standardized framework for treatment of such approaches from an experimental design point of view.

1.2 Related Work

Over the past two decades, several different strategies have been proposed for approximating optimal control laws using learning-based methods. Some of the earlier contributions include the works of Parisini et al. [36] and Åkesson et al. [2], both which consider, albeit different, neural network implementations for non-linear, receding-horizon control problems. More specifically, Parisini et al. restrict their attention to the implementation of a two-layered, sigmoidal feed-forward network for learning an optimal control strategy, whereas Åkesson et al. propose a method for training a network to instead minimize the control cost directly, thereby escaping the need for computing the optimal MPC control signals offline.

Following the advances in various machine learning areas, neural network designs have since then evolved to incorporate more complex features. Examples of such include the activation function Rectified Linear Unit (ReLU), the Long Short-Term Memory (LSTM) structure and the two optimization layers OptNet [4] and cxvpylayers [1]. To some extent, these features each hold some properties akin to those of the MPC and eMPC, and have as such been explored in more recent work on learning-based optimal control.

One method introduced by Chen et al. [10] in their work on learning the eMPC, makes use of a reinforcement learning technique to train a ReLU-based deep neural network for approximating the control law. They note that by choosing the ReLU as activation, the integrated network will represent a PWA function that overlaps with the structure of the eMPC, and thus makes for an attractive choice for synthesizing the controller.

Another key insight of their approach is that they are able to guarantee feasibility, i.e. constraint satisfaction, of the generated control inputs by incorporating an orthogonal projection operation in their network. They enforce this strategy by using Dykstra's projection algorithm to project all network outputs onto a safe region that they define as the conjunction between the maximal control invariant set and the set of input constraints.

The same question is addressed by Maddalena et al. in [31], who take a slightly different approach. Unlike Chen et al., they do not make use of the system model in their method. Instead, they incorporate an implicit parametric quadratic program layer[1] in their network setting and prove that, in combination with two linear layers, it is able to capture the structure of

[1]An OptNet instance.

any *linear* MPC problem. They also show that the resulting closed-loop system can be certified for stability a posteriori.

An interesting factor to consider when learning the MPC is the temporal dependency between the control actions. The ability to model sequential data is an inherent trait of a recurrent neural network and is the reason why such models have already been applied in a variety of MPC settings. Two examples are the works by Spielberg et al. [26] and Lanzetti et al. [29], which both employ recurrent network models with an LSTM structure to synthesize MPC controllers.

1.3 Problem Formulation

As previously discussed, the goal of any learning approach for solving MPC problems is to obtain a meaningful mapping $\mu(\mathbf{x})$ such that for any given initial state $\mathbf{x}$, it produces the control law

$$\mathbf{u} = \mu(\mathbf{x}),$$

where $\mathbf{u}$ is the solution to the MPC. In order for one to make a consistent characterization of any learning approach, two important aspects have to be analyzed:

- The nature of the mapping,
- The nature of data, in the context of training and evaluation.

Though learning approaches to MPC are not new, a framework for such a characterization of learning approaches in the MPC setting does not exist in literature. This is what forms the motivating force behind this thesis. The objective is to investigate and compare different neural network-based MPC approaches, and to ultimately arrive at general performance trends that will help decide between the different possible modes of operation.

1.4 Contributions

The main contribution of this thesis is a framework for consistent analysis of learning-based approaches for MPC in terms of the following key aspects:

- *Structure of the neural network*: This work examines how to train a neural network for implementing the MPC, using the state as input and the control law as output. Several different architectures will be discussed, ranging from a black box neural network to architectures which are more structure-aware in terms of both the MPC problem as well as the unconstrained LQR.
- *Nature of the data generation*: It is not obvious how to efficiently generate training data when learning the control law as a mapping. Grid-based approaches for sampling the input space would work for low state dimensions, but become cumbersome for high-dimensional systems. With this in mind, this thesis studies the use of an efficient and statistically motivated Hit-and-Run sampler that scales well to high dimensions.
- *Evaluation of performance*: It generally remains unclear as to how different network structures can be evaluated consistently. This project studies how the different neural network approaches compare in terms of two different metrics.

1.5 Outline of Thesis

Chapter 2 will introduce the reader to constrained linear systems as well as some concepts from invariance set theory necessary for understanding the implemented network structures.

Chapter 3 provides an overview of the different control principles considered in this work. More specifically, it treats the LQR, the MPC and the eMPC.

Chapter 4 gives a theoretical background on the basics of artificial neural networks necessary for understanding this thesis. In particular, it covers the neuron, activation functions, network architectures, the supervised learning paradigm, and, lastly, the convex optimization layer.

Chapter 5 is devoted to the proposed network approaches. It is mainly concerned with the motivation behind the design of the different structures and how they relate to the original problem of learning an MPC control law.

Chapter 6 describes the training procedure and introduces the performance metrics used for evaluating the networks. It also presents the algorithm used for generating the training and test data.

Chapter 7 presents the results from the network evaluations. It describes in detail the performed experiments and the systems they were conducted on.

Chapter 8 discusses the results from Chapter 7 and summarizes the project. It also includes suggestions on possible future work.

Chapter 2

Constrained Linear Systems

This chapter serves as a gentle introduction to constrained linear discrete-time systems and their structural properties. It also includes a brief section on set invariance, which formally introduces the concept of control invariant sets.

2.1 Modeling of Discrete-Time Systems

A linear, time-invariant discrete-time system can be described by the difference equations

$$\begin{aligned} \mathbf{x}_{k+1} &= \mathbf{A}\mathbf{x}_k + \mathbf{B}\mathbf{u}_k \\ \mathbf{y}_k &= \mathbf{C}\mathbf{x}_k + \mathbf{D}\mathbf{u}_k \end{aligned} \tag{2.1}$$

where $\mathbf{x}_k \in \mathbb{R}^n$ is the state vector of the system at time $k \in \mathbb{Z}^+$, $\mathbf{u}_k \in \mathbb{R}^m$ the input vector, $\mathbf{y}_k \in \mathbb{R}^p$ the output vector, $\mathbf{A} \in \mathbb{R}^{n \times n}$ the dynamics matrix, $\mathbf{B} \in \mathbb{R}^{n \times m}$ the control matrix, $\mathbf{C} \in \mathbb{R}^{p \times n}$ the sensor matrix, and $\mathbf{D} \in \mathbb{R}^{p \times m}$ the direct term (often set to 0) [5].

While such a model provides a good understanding of the underlying dynamics of a system, it is generally unable to capture the full complexity of most physical processes. For this reason, there will always be some deviation between the mathematical model in (2.1) and the real system it is representing. Exploiting feedback in the system control design is an important means for dealing with this uncertainty.

Most real-life systems are in addition subject to constraints on both the state and the input, defined on their most general form by

$$\mathbf{x}_k \in \mathcal{X}, \quad \mathbf{u}_k \in \mathcal{U}, \quad \forall\, k \in \mathbb{Z}^+ \tag{2.2}$$

where the sets $\mathcal{X} \subseteq \mathbb{R}^n$ and $\mathcal{U} \subseteq \mathbb{R}^m$ are polyhedra. The state constraints are generally imposed by safety and performance specifications or other behavioural requirements, while the input constraints, on the other hand, are characterized by actual physical limitations — meaning they will be enforced whether the synthesized controller manages to satisfy them or not. Handling constraints thus becomes an important aspect of regulator design; not only for safety reasons, but also because it generally holds that operating near constraint boundaries increases performance [17, 24, 38].

2.2 Structural Properties

The structural properties of a system are key concepts for understanding the synthesis of controllers with observers and state feedback. They describe how the applied input affects the state vector as well as how the latter shows in the output, thereby providing means of quantifying both the ability to induce certain behaviour in the system as well as the ability to reconstruct the state from output measurements [5, 16]. Below follows the definitions[1] of the structural properties of a system of the form (2.1).

Definition 1 (Controllability)**.** A system is said to be *controllable*, if for any pair of states $(\mathbf{x}, \mathbf{z})$ there exists an input sequence $\{\mathbf{u}_0, \mathbf{u}_1, \dots, \mathbf{u}_{T-1}\}$ such that $\mathbf{z}$ can be reached from $\mathbf{x}$ in a finite time T.

[1]Definitions adapted from [5, 28, 38].

Definition 2 (Observability)**.** A system is said to be *observable*, if there exists a finite T such that the state $\mathbf{x}_T$ can be uniquely determined through the measurements of the input and output sequences $\{\mathbf{u}_0, \mathbf{u}_1, \dots, \mathbf{u}_{T-1}\}$ and $\{\mathbf{y}_0, \mathbf{y}_1, \dots, \mathbf{y}_{T-1}\}$.

A powerful tool for testing a system for controllability and observability is the Hautus lemma for discrete time systems.

Lemma 2.2.1. *A system is controllable, if and only if*

$$\operatorname{rank}\begin{bmatrix}\lambda\mathbf{I}-\mathbf{A} & \mathbf{B}\end{bmatrix} = n, \quad \forall\ \lambda \in \operatorname{eig}(\mathbf{A}).$$

Lemma 2.2.2. *A system is observable, if and only if*

$$\operatorname{rank}\begin{bmatrix}\lambda\mathbf{I}-\mathbf{A} \\ \mathbf{C}\end{bmatrix} = n, \quad \forall\ \lambda \in \operatorname{eig}(\mathbf{A}).$$

For a controllable system it holds that all eigenvalues (modes) of $\mathbf{A}$ can be modified by applying state feedback. Similarly, for an observable system it holds that all modes of $\mathbf{A}$ can be modified by applying output feedback. For uncontrollable (unobservable) systems, there are modes which cannot be modified using feedback. These are referred to as uncontrollable (unobservable) modes, and it is of interest to study their stability. The following weaker notions are therefore useful [16, 38, 41].

Definition 3 (Stabilizability, Detectability)**.** A system is *stabilizable*, if its uncontrollable modes lie within the stability region. It is said to be *detectable*, if its unobservable modes lie within the stability region.

The Hautus conditions for stabilizability and detectability are as follows.

Lemma 2.2.3. *A discrete-time system is stabilizable, if and only if*

$$\operatorname{rank}\begin{bmatrix}\lambda\mathbf{I}-\mathbf{A} & \mathbf{B}\end{bmatrix} = n, \quad \forall\ |\lambda| \geq 1. \tag{2.3}$$

Lemma 2.2.4. *A discrete-time system is detectable, if and only if*

$$\operatorname{rank}\begin{bmatrix}\lambda\mathbf{I}-\mathbf{A} \\ \mathbf{C}\end{bmatrix} = n, \quad \forall\ |\lambda| \geq 1. \tag{2.4}$$

It is easy to see that observability implies stabilizability, just as observability implies detectability. The reverse implications, however, do not hold.

2.3 Set Invariance Theory

The concept of set invariance is closely related with safety and feasibility of control systems. In fact, the existence of control invariant sets is often a fundamental step for solving the control synthesis problem in the presence of constraints [8, 9, 24].

In words, a control invariant set is defined as a set of admissible initial states in the state space for which there exists a control law such that the generated trajectory is kept within the set. More specifically, the set contains all initial states whose trajectories does not violate the system constraints. Below follows the more rigorous definitions from [9].

Definition 4 (Control invariant set). *A set $\mathcal{C} \subset \mathcal{X}$ is said to be a control invariant set for the system* (2.1) *subject to the constraints in* (2.2) *if*

$$\mathbf{x}_k \in \mathcal{X} \implies \exists \mathbf{u}_k \text{ such that } \mathbf{x}_{k+1} \in \mathcal{C}.$$

Definition 5 (Maximal control invariant set). *The set $\mathcal{C}_\infty \subset \mathcal{X}$ is said to be the maximal control invariant set for the system* (2.1) *subject to the* *constraints in* (2.2)*, if it is control invariant and contains all control invariant sets contained in* $\mathcal{X}$*.*

From the definition it follows that the maximal control invariant set is the largest set for which one can expect *any* controller to work [9]. An algorithm for computing $\mathcal{C}_\infty$ can be found in [9]. The procedure is also provided in Algorithm 1 for the reader's convenience[1].

Example 2.3.1. Consider the second order system

$$\mathbf{x}_{k+1} = \mathbf{A}\mathbf{x}_k + \mathbf{B}\mathbf{u}_k = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \mathbf{x}_k + \begin{bmatrix} 2 \\ 1 \end{bmatrix} \mathbf{u}_k \tag{2.5}$$

subject to the constraints

$$\begin{bmatrix} -5 \\ -5 \end{bmatrix} \leq \mathbf{x}_k \leq \begin{bmatrix} 5 \\ 5 \end{bmatrix}, \quad -1 \leq u_k \leq 1, \quad \forall\ k. \tag{2.6}$$

Using Algorithm 1, the related maximal control invariant set is computed as

$$\begin{bmatrix} -0.71 & -0.71 \\ -0.89 & -0.45 \\ 0.71 & 0.71 \\ 0.89 & 0.45 \\ -1 & 0 \\ 0 & -1 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \mathbf{x} \leq \begin{bmatrix} 4.24 \\ 4.02 \\ 2.24 \\ 4.02 \\ 5 \\ 5 \\ 5 \\ 5 \end{bmatrix}. \tag{2.7}$$

The set is also depicted in Figure 2.3.1.

Remark 2.3.1. It is important to note that Algorithm 1 will not necessarily terminate in finite time, and is thus not guaranteed to converge to the maximal control invariant set.

[1]The precursor set is defined as: $\text{Pre}(\mathcal{S}) = \{\mathbf{x} \in \mathbb{R}^n : \exists \mathbf{u} \in \mathcal{U} \text{ s.t. } \mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{u} \in \mathcal{S}\}$.

Algorithm 1 Computation of $\mathcal{C}_\infty$

```
1: procedure COMPUTATION OF $\mathcal{C}_\infty$
2:     $\Omega_0 \leftarrow \mathcal{X}$, $k \leftarrow -1$
3:     repeat
4:         $k \leftarrow k+1$
5:         $\Omega_{k+1} \leftarrow \text{Pre}(\Omega_k) \cap \Omega_k$
6:     until $\Omega_{k+1} = \Omega_k$
7:     $\mathcal{C}_\infty \leftarrow \Omega_{k+1}$
8:     return $\mathcal{C}_\infty$
9: end procedure
```

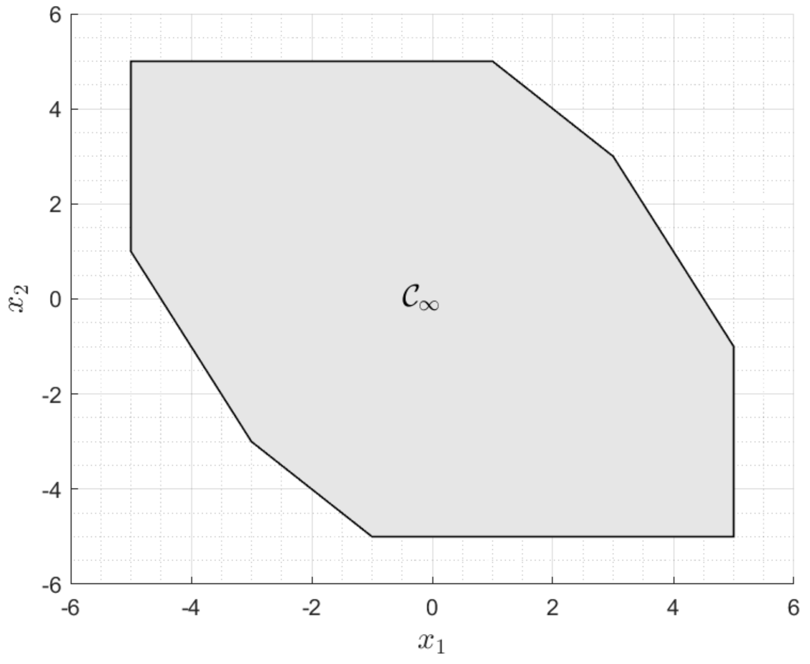


Figure 2.3.1: The maximal control invariant set, $\mathcal{C}_\infty$, for the system in (2.5) subject to the constraints in (2.6).

Chapter 3

Control Principles

This chapter provides an introduction to the control principles considered in this work. It begins with a summary of the unconstrained and constrained LQR, followed by a discussion on the principles of the MPC and how it relates to the LQR. The chapter is concluded with a section on the eMPC and an example of the same.

3.1 Linear Quadratic Optimal Control

Consider a discrete-time linear time-invariant system evolving in times as

$$\mathbf{x}_{k+1} = \mathbf{A}\mathbf{x}_k + \mathbf{B}\mathbf{u}_k \tag{3.1}$$

where, as before, $\mathbf{x}_k \in \mathbb{R}^n$ and $\mathbf{u}_k \in \mathbb{R}^m$ denote the state and input vectors at time k, respectively. Assume that the pair $(\mathbf{A}, \mathbf{B})$ is stabilizable.

The infinite-horizon Linear Quadratic (LQ) problem is then defined as the problem of finding a control input sequence $\{\mathbf{u}_0, \mathbf{u}_1, \ldots, \mathbf{u}_\infty\}$ that will steer the system in (3.1) from some initial state, $\mathbf{x}_0 = \mathbf{x}(0)$, to the origin while minimizing the control cost function defined as

$$J_\infty = \sum_{k=0}^{\infty} (\mathbf{x}_k^\top \mathbf{Q} \mathbf{x}_k + \mathbf{u}_k^\top \mathbf{R} \mathbf{u}_k) \tag{3.2}$$

where $\mathbf{R} \succ 0$ is a symmetric cost matrix penalizing the input, and $\mathbf{Q} \succeq 0$ a symmetric cost matrix penalizing the state error, such that $(\mathbf{Q}^{1/2}, \mathbf{A})$ is detectable. The solution to this problem results in the optimal control law known as the LQR[1] given by

$$\mathbf{u}_k^* = -(\mathbf{B}^\top \mathbf{P}_\infty \mathbf{B} + \mathbf{R})^{-1} \mathbf{B} \mathbf{P}_\infty \mathbf{A} \mathbf{x}_k = -\mathbf{L} \mathbf{x}_k \tag{3.3}$$

where $\mathbf{P}_\infty$ solves the *Discrete Time Algebraic Riccati Equation* [9, 16]

$$\mathbf{P}_\infty = \mathbf{A}^\top \mathbf{P}_\infty \mathbf{A} - \mathbf{A}^\top \mathbf{P}_\infty \mathbf{B} (\mathbf{R} + \mathbf{B}^\top \mathbf{P}_\infty \mathbf{B})^{-1} \mathbf{B}^\top \mathbf{P}_\infty \mathbf{A} + \mathbf{Q}. \tag{3.4}$$

The cost function in (3.2) represents a trade-off between the control action and the control error (distance to the origin). A controller for which $\mathbf{Q}$ is set large relative to $\mathbf{R}$ will prioritize reaching the origin as quickly as possible over reducing the control action, and vice versa. A central aspect of LQR design is choosing appropriate values of $\mathbf{Q}$ and $\mathbf{R}$, also known as tuning, which requires knowledge of the system under study. A common, practical choice is to set the matrices to be diagonal

$$\mathbf{Q} = \begin{bmatrix} q_1 & & 0 \\ & \ddots & \\ 0 & & q_n \end{bmatrix}, \quad \mathbf{R} = \begin{bmatrix} r_1 & & 0 \\ & \ddots & \\ 0 & & r_m \end{bmatrix}.$$

In this way, the diagonal weights will reflect the influence of each individual (squared) state and input on the total cost [5, 38].

The LQR formulation above can be extended to also consider systems subject

[1]Reader is referred to [9, 16] for derivation.

to state and input constraints of the form

$$\mathbf{x}_k \in \mathcal{X}, \quad \mathbf{u}_k \in \mathcal{U}, \tag{3.5}$$

where the sets $\mathcal{X} \subseteq \mathbb{R}^n$ and $\mathcal{U} \subseteq \mathbb{R}^m$ are polyhedra. The problem is then reformulated as

$$\begin{aligned} \underset{\mathbf{u}_{0:\infty}}{\arg\min} \quad & J_\infty = \sum_{k=0}^{\infty} (\mathbf{x}_k^\top \mathbf{Q} \mathbf{x}_k + \mathbf{u}_k^\top \mathbf{R} \mathbf{u}_k) \\ \text{s.t.} \quad & \mathbf{x}_{k+1} = \mathbf{A}\mathbf{x}_k + \mathbf{B}\mathbf{u}_k,\ k = 0, \ldots, \infty \\ & \mathbf{x}_k \in \mathcal{X},\ \mathbf{u}_k \in \mathcal{U},\ k = 0, \ldots, \infty \\ & \mathbf{x}_0 = \mathbf{x}(0) \end{aligned} \tag{3.6}$$

and is referred to as the infinite-horizon Constrained Linear Quadratic Regulation (CLQR) problem.

Due to the infinite number of decision variables and constraints included in the optimization, the CLQR is in general very difficult to solve. There exists some work on how to address the CLQR directly, including e.g. [40], [20] and [42], but one often resorts to approximate methods instead. Model predictive control is the most prevalent approximation scheme and is presented in the section below.

3.2 Model Predictive Control

Model predictive control is an online strategy for designing an infinite-horizon sub-optimal controller in a receding horizon fashion described in the following. At each time instant k, an optimal control input sequence $\{\mathbf{u}_k, \mathbf{u}_{k+1}, \ldots, \mathbf{u}_{k+N}\}$ is computed by solving a constrained optimization problem over a finite-time horizon, N. The first input in the sequence, $\mathbf{u}_k$, is then applied to the system and the same procedure is repeated at the next time instant, $k+1$, based on the evolved state to generate the next control sequence, $\{\mathbf{u}_{k+1}, \mathbf{u}_{k+1}, \ldots, \mathbf{u}_{k+N+1}\}$. Thus, the horizon recedes over time.

The general MPC problem is formulated as

$$\begin{aligned}
\underset{\mathbf{u}_{0:N-1}}{\arg\min} \quad & J(\mathbf{x}_0) = \mathbf{x}_N^\top \mathbf{Q}_N \mathbf{x}_N + \sum_{k=0}^{N-1} (\mathbf{x}_k^\top \mathbf{Q} \mathbf{x}_k + \mathbf{u}_k^\top \mathbf{R} \mathbf{u}_k) \\
\text{s.t.} \quad & \mathbf{x}_{k+1} = \mathbf{A}\mathbf{x}_k + \mathbf{B}\mathbf{u}_k,\ k = 0, \ldots, N-1 \\
& \mathbf{x}_k \in \mathcal{X},\ \mathbf{u}_k \in \mathcal{U},\ k = 0, \ldots, N-1 \\
& \mathbf{x}_N \in \mathcal{X}_f
\end{aligned} \tag{3.7}$$

where $\mathbf{x}_0$ is the current state, N the prediction horizon, $\mathbf{Q}_N = \mathbf{Q}_N^\top \succeq 0$ the terminal cost matrix, and $\mathcal{X}_f$ the terminal state constraints. As with the cost matrices, it is not obvious how the prediction horizon should be chosen. A short horizon is preferred for computational simplicity. However, a longer horizon will in general yield more accurate trajectory predictions.

3.2.1 Explicit Model Predictive Control

One considerable disadvantage of MPC is the computational effort required for solving (3.7) online, which makes MPC challenging for very fast processes [6]. Explicit MPC is a strategy for circumventing this issue by pre-computing the control law offline using multiparametric programming techniques.

Given a polytopic set $\mathcal{X}$, for each $\mathbf{x} \in \mathcal{X}$ the eMPC computes a piecewise affine mapping from $\mathbf{x}$ to $\mathbf{u}$ defined over M regions of $\mathcal{X}$. In this way, the eMPC performs a characterization of the control law akin to a look-up table: the only computation required online is then to determine what region the current state is in [3, 6].

Unlike the control law of the MPC, the control law of the eMPC becomes explicitly dependent on $\mathbf{x}$ according to

$$\mathbf{u}(\mathbf{x}) = \begin{cases} \mathbf{F}_1\mathbf{x} + \mathbf{g}_1, & \text{if } \mathbf{H}_1\mathbf{x} \leq \mathbf{k}_1 \\ \mathbf{F}_2\mathbf{x} + \mathbf{g}_2, & \text{if } \mathbf{H}_2\mathbf{x} \leq \mathbf{k}_2 \\ & \vdots \\ \mathbf{F}_M\mathbf{x} + \mathbf{g}_M, & \text{if } \mathbf{H}_M\mathbf{x} \leq \mathbf{k}_M \end{cases} \tag{3.8}$$

where $\mathbf{F}_i$ and $\mathbf{g}_i$ are the parameters of the local functions, and $\mathbf{H}_i\mathbf{x} \leq \mathbf{k}_i$ denote the expressions that characterize the different regions of $\mathcal{X}$.

While the eMPC on the one hand reduces the computational burden in terms of solving for different states, it typically does not scale well with the dimension of the state due to nature of the function (3.8). The complexity of the solution further increases with the number of regions, M, as this is what dictates the amount of memory required for storing the functions in (3.7) [6, 27].

Example 3.2.1. Consider again the double integrator in (2.5) subject to the constraints in (2.6). Let the cost parameters in (3.7) be defined as $\mathbf{Q} = \mathbf{I}$, $\mathbf{R} = 10$, and $\mathbf{Q}_N = \mathbf{I}$, the horizon as $N = 3$, and the terminal constraints as $\mathcal{X}_f = \mathbb{R}^2$. Using the MPT3 [22] toolbox in Matlab to construct the eMPC, one obtains the controller that consists of the seven regions plotted in Figure 3.2.1. The corresponding feedback law (3.8) is plotted in Figure 3.2.2.

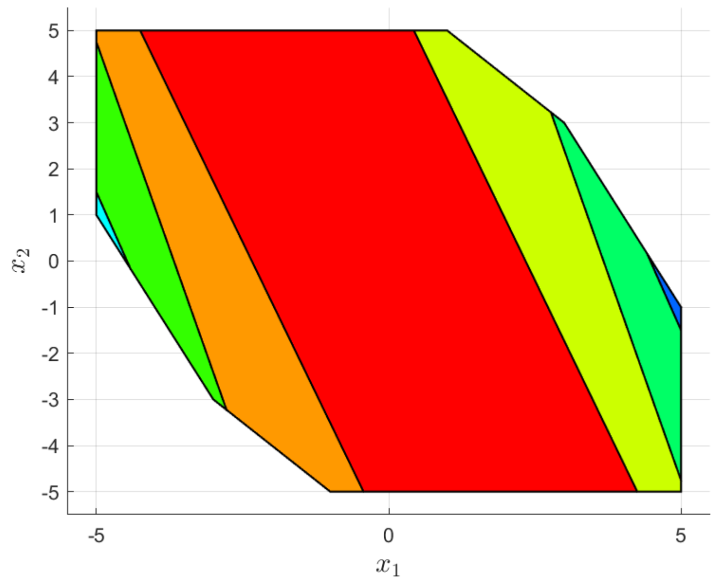


Figure 3.2.1: The seven regions of the eMPC computed in Example 3.2.1.

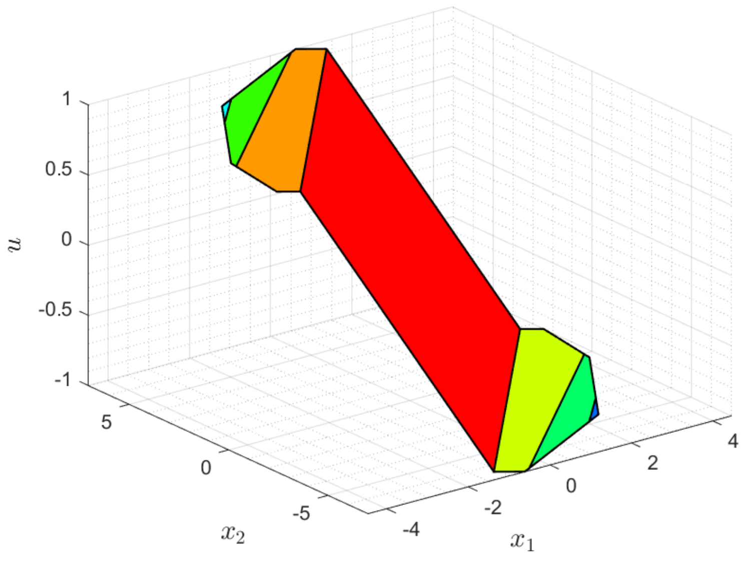


Figure 3.2.2: The PWA feedback law computed in Example 3.2.1.

Chapter 4

Artificial Neural Networks

An artificial neural network is a computational system motivated by the functioning of the biological neural networks that constitute the human brain. A biological neural network is essentially an information-processing unit organized by millions of interconnected neurons (nerve cells), each in itself a complex signaling system capable of inducing or inhibiting activity in other neurons. Such neural events are what enables the brain to perform complex functions and computations, and also what inspired the early development of artificial neural networks [21, 33, 39].

This chapter aims at introducing the reader to the basic concepts of neural networks[1] necessary for understanding this work. It will start off by describing the neurons, the computing elements of the network, and then move on to discuss different network structures and the supervised learning paradigm. The chapter ends with an introduction to convex optimization layers, a state-of-the-art network architecture for integrating differentiable optimization programs in the network structure.

[1]All usages of the terms "neuron" and "neural network" will for the remainder of this report refer to that of the artificial neuron and artificial neural network, respectively.

4.1 The Neuron

The fundamental building blocks of a neural network are the processing units known as neurons. A neuron represents a mathematical function that generates an output signal based on inputs received from other neurons in the network. The neurons communicate by transmitting signals across their connecting links, sometimes also referred to as synapses. Each such connection is assigned a weight coefficient, w. In mathematical terms, this implies that the signal going from a neuron i to a neuron j will be multiplied by the weight w_{ij} of the synapse connecting the two.

The operation realized by the neuron is given by

$$y_j = \varphi\left(\sum_{i=1}^{l} w_{ij}x_i + b_i\right) \tag{4.1}$$

where $\{x_1, x_2, \ldots, x_l\}$ are the input signals to the j:th neuron, $\{w_{1l}, w_{2l}, \ldots, w_{lj}\}$ the link weights, b_j the bias, $\varphi(\cdot)$ the activation function, and y_j the generated output. Together, the weights and biases form the learnable parameters that are adjusted by the network during training.

A visual representation of the neuron is provided in Figure 4.1.1.

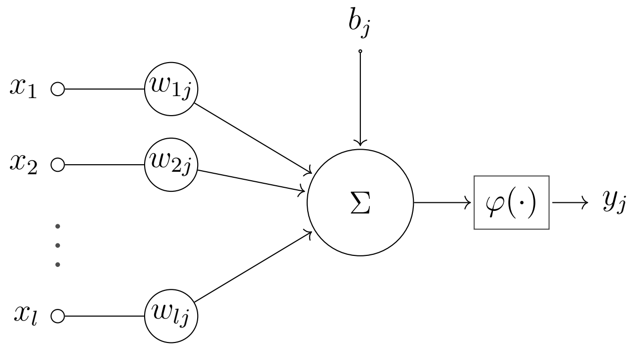


Figure 4.1.1: Visual representation of a neuron. The link weights are labeled with w and the bias with b. The activation function is denoted by $\varphi(\cdot)$. Figure adapted from [21].

4.1.1 Activation Functions

The activation function determines the output of a neuron and plays a fundamental role in the neural network. The non-linearity in the form of activation functions appearing in multiple is key to the remarkable performance of neural networks in many real-world tasks. From (4.1), it is easy to see that by omitting the activation function, the neural operation is reduced to that of a linear transformation. The same happens when the activation function is chosen to be linear. The common practice is therefore to employ non-linear activation functions, as these enable the integrated network to learn and process more complex data patterns [7, 21, 39].

In addition to being non-linear, an activation function should also be differentiable and monotonic. This is because training of neural networks typically proceeds in form of backpropagation, which uses gradients of the network parameters. Popular choices include the logistic sigmoid function, tanh and the ReLU plotted in Figures 4.1.2 and 4.1.3.

The sigmoid is defined as

$$\sigma(x) = \frac{1}{1+e^{-x}} \tag{4.2}$$

and is typically used in probabilistic classifiers, as it restricts its outputs to lie in the range $[0,1]$. The tanh relates to the sigmoid by $\tanh(x) = 2\sigma(2x) - 1$ and has the mathematical form

$$\tanh(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}. \tag{4.3}$$

Both the sigmoid and the tanh are saturating functions. As shows in Figure 4.1.3, this implies that their gradients will approach zero for very small and very large values of x. In gradient-based learning, where the parameter update is proportional to the gradient of some error or loss function, this phenomenon may induce an issue known as the vanishing gradient problem.

Should it occur, it may prevent the weights from updating and could thereby stop the network from learning altogether [19, 35, 46].

The ReLU function defined as

$$\mathrm{ReLU}(x) = \max(0, x) = \begin{cases} x, & \text{if } x \geq 0 \\ 0, & x < 0 \end{cases} \tag{4.4}$$

does not suffer from this issue. Its gradient will, however, evaluate to zero for negative values of x, causing the ReLU to "die" and output zero (0) for all future inputs[1]. Nevertheless, because the ReLU function is unbounded, the set of inputs for which the gradient is nonzero is still larger compared to those of the tanh and sigmoid. Further, whenever the gradient is nonzero for the ReLU, it is always one, whereas for the sigmoid and tanh, the gradient lies in the range $[0, 1]$. This adds to the computational efficiency of the ReLU.

Another advantage of the ReLU activation is its computational simplicity. As opposed to the sigmoid and tanh, the ReLU does not involve any exponential operations in its computation, which makes it significantly more efficient in comparison. Because of this, the ReLU has become the most popular choice of activation function in most network-based applications [13, 14, 46]. It is worth noting, however, that the ReLU is an unsuitable choice for the output layer for regression problems, as it only produces non-negative values.

4.2 Network Structures

In its most general form, a neural network is a weighted, directed graph comprised of neurons ordered into different layers. Depending on the interconnection pattern of these layers, one distinguishes between the two main network structures (architectures): Feed-Forward Neural Networks

[1]This is known as the dying ReLU problem and can be avoided by implementing a leaky ReLU.

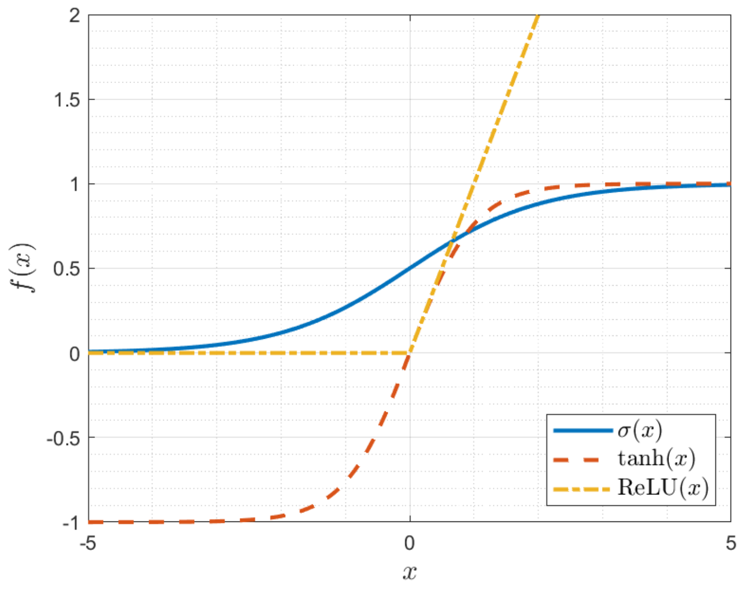


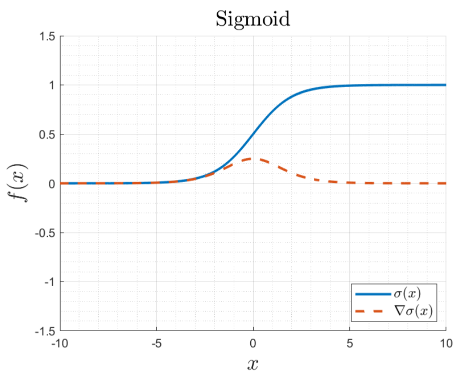
Figure 4.1.2: Comparison of the three activation functions sigmoid, tanh and ReLU.

(FFNN) and Recurrent Neural Networks (RNN).

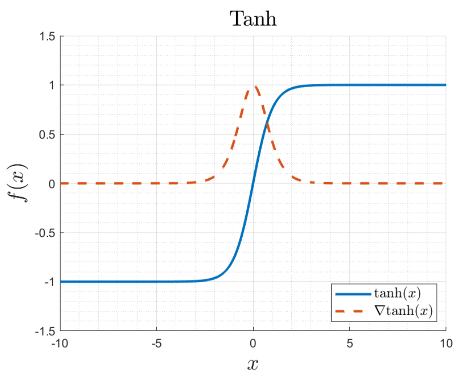
4.2.1 Feed-Forward Neural Networks

In a feed-forward neural network, the layers are sequentially connected without any feedback loops. Each neuron of a particular layer is unidirectionally connected to the neurons of the subsequent layer, thereby restricting the information to flow only in the forward direction. The simplest form of a feed-forward network is the single-layer FFNN consisting only of an input layer and an output layer. The designation “single-layer” stems from the input layer not being considered as a layer, as it merely feeds the inputs to the output layer without performing any computation. All processing is thus limited to the output layer.

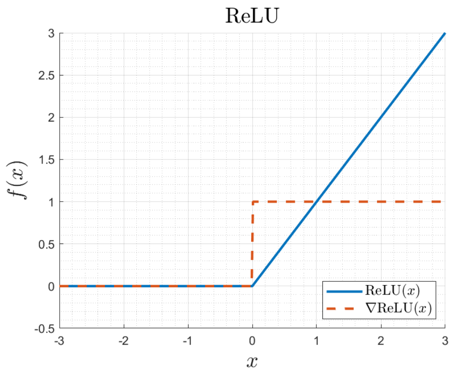
A more complex feed-forward network is the multi-layer FFNN, which



(a)



(b)



(c)

Figure 4.1.3: The three activation functions and their respective gradients.

incorporates one or more hidden layers in its structure. The hidden layers are inserted between the input and output layers to induce nonlinearities in the network, which in turn enables the network to extract higher order features from data [7, 21, 39]. It is known that with a low number of layers the performance typically saturates after a point, and that increasing the number of layers often adds to the performance. The general structure of an FFNN is shown in Figure 4.2.1.

Because of the acyclic structure of a feed-forward network, it is possible to derive a closed-form expression for its learnt mapping, $\mu(\mathbf{x})$, that is uniquely defined by the network structure. Recall the neural operation in (4.1). For a fully connected network it then follows that

$$\mu(\mathbf{x}) = \varphi(\mathbf{W}_L \varphi(\mathbf{W}_{L-1} \varphi(\cdots \varphi(\mathbf{W}_1 \mathbf{x} + \mathbf{b}_1) \cdots) + \mathbf{b}_L) \tag{4.5}$$

where $\Phi(\cdot)$ denotes the point-wise activation function, $\{\mathbf{W}_i, \mathbf{b}_i\}_{i=0}^{L}$ the parameters (weights and biases) adjusted by the network during training, and L the number of layers in the network.

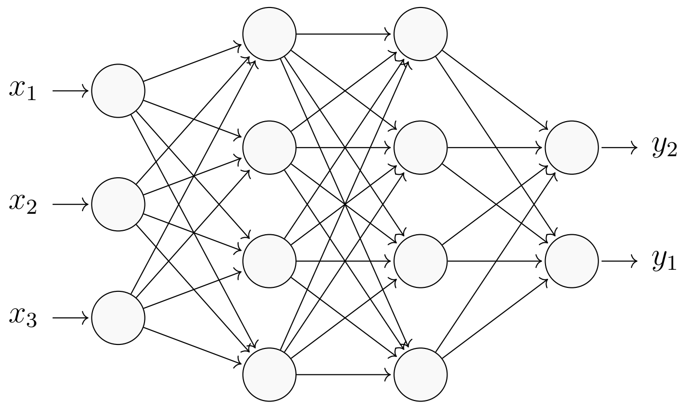


Figure 4.2.1: An example of a fully connected FFNN with two hidden layers. The weights and biases have been omitted for simplicity.

4.2.2 Recurrent Neural Networks

A recurrent neural network is characterized by the presence of at least one cycle in its structure. Cycles occur because of feedback connections, which may be induced either

- *locally*, which refers to feedback within the hidden layer, or
- *globally*, which refers to feedback from the output layer to the input layer.

Both types are illustrated in the RNN schematic in Figure 4.2.2.

The introduction of feedback loops makes it possible for the network to reuse and temporarily store the results of previous computations. This adds a

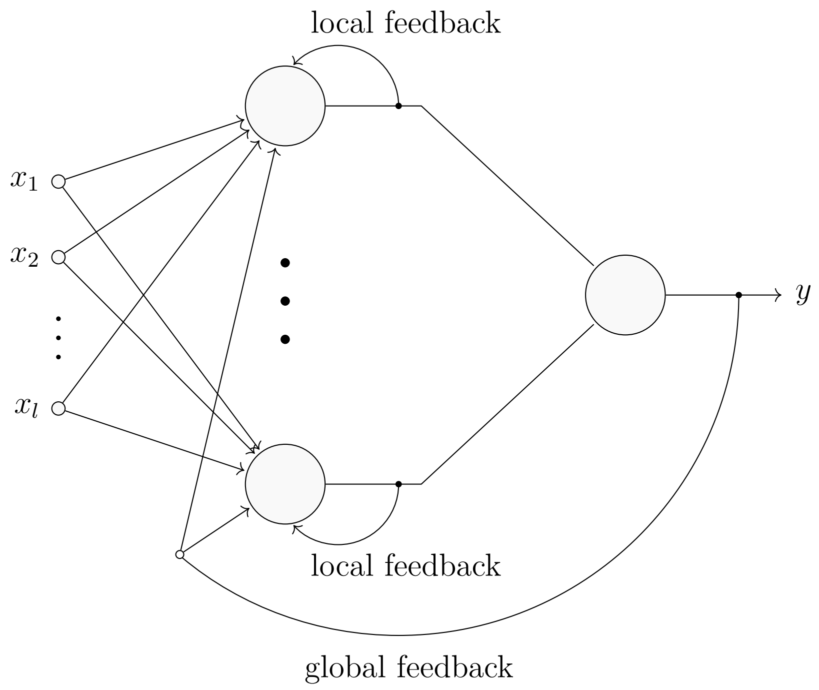


Figure 4.2.2: A recurrent neural network with both local and global feedback. Figure adapted from [32].

temporal dimension to the network's structure, which is of particular interest for applications dealing with sequential data, such as time series forecasting and speech recognition.

An RNN with only local feedback is typically modeled as in Figure 4.2.3. For easy visualization of the forward pass, it is convenient to unroll (or unfold) the RNN in time to obtain the deep feed-forward structure in Figure 4.2.4. It is important to note that the layers in the unfolded representation reuse the same weights at every timestep — only the outputs and internal states changes [19, 30].

Similarly to the FFNN, a serious flaw of the basic RNN is that it suffers from the vanishing gradient problem. In practice, the vanishing gradients greatly limit the network's capability of processing contextual information in the sense that they prevent the network from learning high temporal distance relationships between data. For the interested reader, [19] provides a nice illustration of this. The LSTM is a type of RNN designed to overcome this issue.

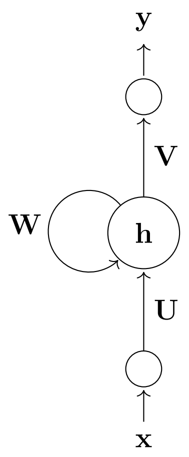


Figure 4.2.3: Model of an RNN with only local feedback. Each node represents a layer of neurons. The weights **U**, **V**, and **W** denote the input weights, the output weights and the recurrent weights, respecitvely. Figure adapated from [30].

4.2.3 Long Short-Term Memory

An LSTM network is composed of a set of linked blocks referred to as LSTM cells or LSTM units. Each unit contains a memory cell (cell state) and the three gating elements: the input, output, and forget gates, which represent the read, write, and reset operations of the cell state, respectively. Each gate consists of a sigmoid feed-forward layer followed by a pointwise multiplication operation, as shown in Figure 4.2.5. A graphical representation of the entire LSTM cell is provided in Figure 4.2.6, in which the elements are

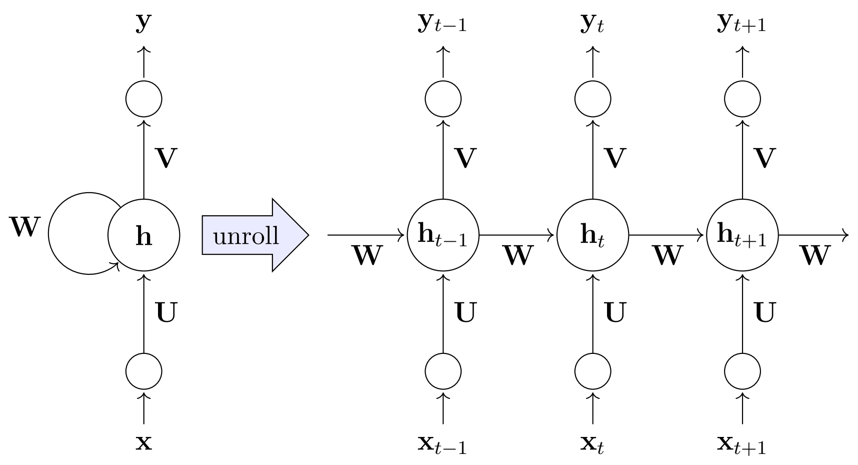


Figure 4.2.4: An unrolled RNN. Each node represents a layer of network units. The hidden nodes are labeled **h**. The weights **U**, **V**, and **W** denote the input weights, output weights, and the recurrent weights, respectively. Note that the same weights are used at each time step. Figure adapted from [19].

mathematically defined by

$$\begin{aligned}
\mathbf{f}_t &= \sigma\left(\mathbf{W}_f \mathbf{h}_{t-1} + \mathbf{U}_f \mathbf{x}_t + \mathbf{b}_f\right), \\
\mathbf{i}_t &= \sigma\left(\mathbf{W}_i \mathbf{h}_{t-1} + \mathbf{U}_i \mathbf{x}_t + \mathbf{b}_i\right), \\
\mathbf{o}_t &= \sigma\left(\mathbf{W}_o \mathbf{h}_{t-1} + \mathbf{U}_o \mathbf{x}_t + \mathbf{b}_o\right), \\
\tilde{\mathbf{C}}_t &= \tanh\left(\mathbf{W}_C \mathbf{h}_{t-1} + \mathbf{U}_C \mathbf{x}_t + \mathbf{b}_C\right), \\
\mathbf{C}_t &= \mathbf{f}_t \odot \mathbf{C}_{t-1} + \mathbf{i}_t \odot \tilde{\mathbf{C}}_t, \\
\mathbf{h}_t &= \mathbf{o}_t \odot \tanh\left(\mathbf{C}_t\right)
\end{aligned} \tag{4.6}$$

where $\sigma(\cdot)$ denotes the sigmoid activation function, $\tilde{\mathbf{C}}_t$ denote the new candidate values of the cell state, and $\mathbf{b}_f$, $\mathbf{b}_i$, and $\mathbf{b}_o$ the biases.

The purpose of the gates is to regulate the information flow to the memory cell of the LSTM unit. The input gate can be seen as an activation threshold that determines if and to what degree the new input signals should influence the current cell state. If the gate is closed, i.e. has an activation near

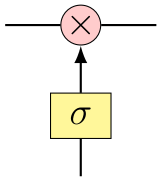


Figure 4.2.5: A graphical representation of an LSTM gate, where σ denotes the sigmoid acitvation function.

0, the cell state remains unchanged. In a similar fashion, the output gate determines how the cell state influences the output of the unit. It is easy to see that by closing the input gate, it is possible to store the information of the memory cell within the unit. The stored information can then further on be accessed by the network at a later time instant, by opening the output gate. This ability to store and access information indefinitely is what uniquely characterizes the LSTM and also what resolves the issue of vanishing gradients [19, 23].

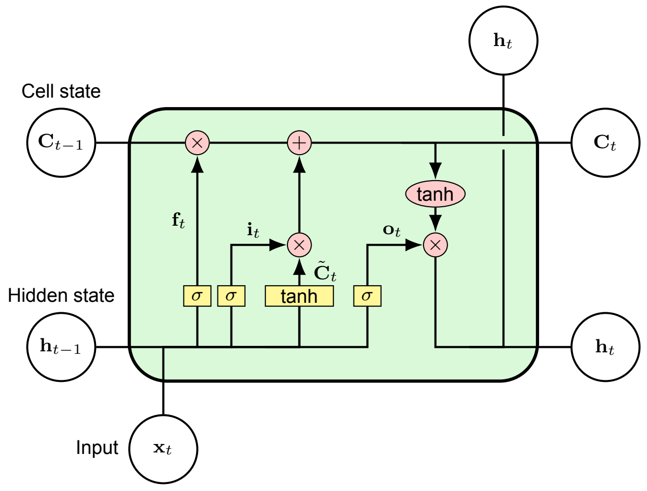


Figure 4.2.6: An LSTM cell. The inputs to the cell consists of the input, $\mathbf{x}_t$, the previous hidden state $\mathbf{h}_{t-1}$, and the previous cell state, $\mathbf{C}_{t-1}$. The cell outputs the updated cell state, $\mathbf{C}_t$, and the new hidden state, $\mathbf{h}_t$. The interim variables $\mathbf{f}_t$, $\mathbf{i}_t$, and $\mathbf{o}_t$ are defined in (4.6). The biases have been omitted for simplicity.

4.3 Supervised Learning

Supervised learning is the most common form of learning and refers to paradigms in which the network is provided a set of input-output pairs to train on. For a given input from the training set, the network generates an output to be compared with the known desired output value, using an error metric. This error metric, or loss function, averaged over all the training samples is uesd to train the network model: the network parameters are chosen to minimize this loss function.

4.3.1 Backpropagation Algorithm

The backpropagation algorithm is a technique used in supervised learning of feed-forward networks that utilizes a gradient descent method to minimize a loss function with respect to the network's parameters. For simplicity, this section will only provide a summarized version of the procedure, which can be found in Algorithm 2. The interested reader is referred to [7] or [39] for the full mathematical derivation.

The equivalence of backpropagation for recurrent neural networks is the backpropagation through time technique, which has been omitted in this section for simplicity. The interested reader can find the procedure in e.g. [19].

Algorithm 2 Backpropagation algorithm

```
1: repeat
2:     for $(\mathbf{x}_i^*, \mathbf{y}_i^*)$ in the training set $\{(\mathbf{x}_1^*, \mathbf{y}_1^*), \dots, (\mathbf{x}_n^*, \mathbf{y}_n^*)\}$ do
3:         Apply $\mathbf{x}_i^*$ to the network,
4:         Propagate $\mathbf{x}_i^*$ through the network to obtain the
           output $\mathbf{y}_i = \mu(\mathbf{x}_i^*)$,
5:         Calculate the loss $\mathcal{L}_i$, by comparing $\mathbf{y}_i$ to $\mathbf{y}_1^*$,
6:         Backpropagate $\mathcal{L}_i$ by calculating the gradients $\nabla\mathcal{L}_i$ w.r.t.
           all weights and biases,
7:         Update weights and biases to minimize $\mathcal{L}_i$.
8:     end for
9: until Until error is below a specified threshold.
```

4.4 Differentiable Convex Optimization Layers

There are cases where one comes across convex objectives being part of the neural network chain. In such cases, in order to update the neural network, one must also compute gradients of the convex problem. As a result, there is a need to have convex optimization problems to be cast as differentiable problems so that backropagation can be applied.

Recent work on neural network architectures has introduced a framework for embedding differentiable optimization problems as individual layers within a deep, feed-forward network structure as a means for introducing domain-specific knowledge to the network structure. This section will give a brief introduction to the cvxpylayers [1], a Python library for implementing differentiable convex opitmization layers using CVXPY [15], which is a Python-embedded modeling language for convex optimization.

The cvxpylayers requires the optimization program to be constructed by the grammar Disciplined Parametrized Programming (DPP) introduced in [**1**], which can be viewed as a subset of the modeling methodology disciplined convex optimization used for constructing convex optimization models. The

DPP consists of a collection of functions with known curvature (affine, convex or concave) and per-argument monotonicities, and a composition ruleset that will guarantee the convexity of the constructed program [1, 18].

Using DPP to construct the expressions, a disciplined parametrized program can then be formulated as an optimization problem of the form

$$\begin{aligned} \arg\min_{\mathbf{x}} \quad & f(\mathbf{x}, \theta) \\ \text{s.t.} \quad & g_i(\mathbf{x}, \theta) \leq \tilde{g}_i(\mathbf{x}, \theta), \quad i = 1, \dots, m_1 \\ & h_j(\mathbf{x}, \theta) = \tilde{h}_j(\mathbf{x}, \theta), \quad j = 1, \dots, m_2 \end{aligned} \tag{4.7}$$

where $\mathbf{x} \in \mathbb{R}^n$ is a variable, $\theta \in \mathbb{R}^p$ is a learnable parameter vector, the g_i are convex, $\tilde{g}_j$ are concave, and h_i and $\tilde{h}_i$ are affine. The parameter θ is learned to minimize some scalar function of the solution, $\mathbf{x}^*$, of (4.7).

The cvxpylayer solves the problem in (4.7) by first canonicalizing it into a cone program of the form

$$\begin{aligned} \text{minimize} \quad & \mathbf{c}^T\mathbf{x} \\ \text{s.t.} \quad & \mathbf{b} - \mathbf{A}\mathbf{x} \in \mathcal{K} \end{aligned} \tag{4.8}$$

where $\mathbf{x} \in \mathbb{R}^n$ is the variable, $\mathcal{K}$ a non-empty, closed, convex cone, and $\mathbf{A}$, $\mathbf{b}$ and $\mathbf{c}$ are the matrices and vectors of appropriate sizes denoting the cone problem data. Next, it calls a cone solver, s, to obtain the solution

$$\tilde{\mathbf{x}}^* = s(\mathbf{A}, \mathbf{b}, \mathbf{c}) \tag{4.9}$$

which is mapped to a solution $\mathbf{x}^*$ of the original problem. The solution map $\mathcal{S} : \mathbb{R}^p \to \mathbb{R}^n$ can thus be expressed as

$$\mathcal{S} = \mathbf{R} \circ s \circ \mathbf{C} \tag{4.10}$$

where $\mathbf{C}$ is the canonicalizer, s the cone solver, and $\mathbf{R}$ the retriever that maps (4.9) to $\mathbf{x}^*$. The derivation of the derivative of (4.10) required for the backpropagation algorithm has been left out for simplicity, but can be found in [1].

Chapter 5

Proposed Neural Network Approaches

This chapter provides an overview of the different neural network architectures investigated in this project. Three different FFNN structures and one RNN structure have been implemented using PyTorch [37] in Python.

The feed-forward networks have been designed to at each time instant, k, take the state vector $\mathbf{x}_k$ as input and produce the corresponding output mapping $\mathbf{u}_k = \mu(\mathbf{x}_k, \theta)$. The recurrent network takes as input a sequence of states of length l, $\{\mathbf{x}_i\}_{i=k-l}^{k}$, and produces the corresponding output $\mathbf{u}_k = \mu(\{\mathbf{x}_i\}_{i=k-l}^{k}, \theta)$. In both mappings, θ denotes the learnable parameters of the networks.

5.1 Black Box Neural Network

The Black Box Neural Network (BBNN) refers to a network that is completely agnostic to any MPC specific information. This means that the learning process relies solely on the input-output samples without explicitly considering the system's dynamics or the state and input constraints. The network consists of one input layer of width $n = \dim(\mathbf{x})$, two hidden layers of width eight, and a final output layer of width $m = \dim(\mathbf{u})$. A schematic of this structure is shown in Figure 5.1.1.

As was noted in Section 1.2, the use of ReLU as the activation function has been shown to be well motivated in the MPC setting, primarily due to its PWA nature [10]. It was, however, pointed out in Section 4.1.1 that the ReLU makes for an unsuitable choice of output activation for regression [14]. Therefore, only the hidden layers employ the ReLU activation. The output layer is instead assigned the linear identity activation function defined as

$$f(x) = x$$

to allow for any negative values of $\mathbf{u}$ and to avoid any unnecessary saturation of the output.

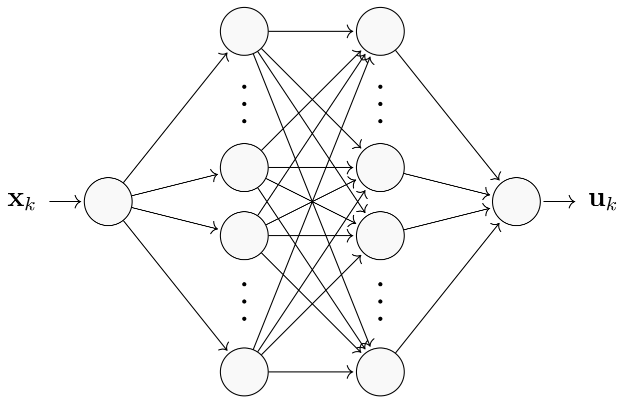


Figure 5.1.1: Schematic of the three-layer BBNN.

5.2 Projection Neural Network

As is also discussed in [10], the basic FFNN ReLU structure alone does not come with any guarantees on constraints satisfaction when employed in an MPC setting. In [10], the suggested approach for obtaining such (guarantees) entails extending the network by using a projection algorithm that utilizes knowledge of the MPC structure to ensure the feasibility of the subsequent state and control trajectories. Following the same line of thought, a similar projection strategy is investigated in this work by implementing a neural network that incorporates a state-of-the-art differentiable convex optimization layer as the final layer in its structure. This network introduces MPC problem specific information to the BBNN structure described in the previous section, and will be referred to as the Projection Neural Network (PNN). The idea of the PNN is to let the optimization layer project the output, $\tilde{\mathbf{u}}_k$, following the BBNN block onto a feasible region $\mathcal{R}(\mathbf{x}_k)$, which will be defined later in this section. A schematic of the PNN structure can be found in Figure 5.2.1.

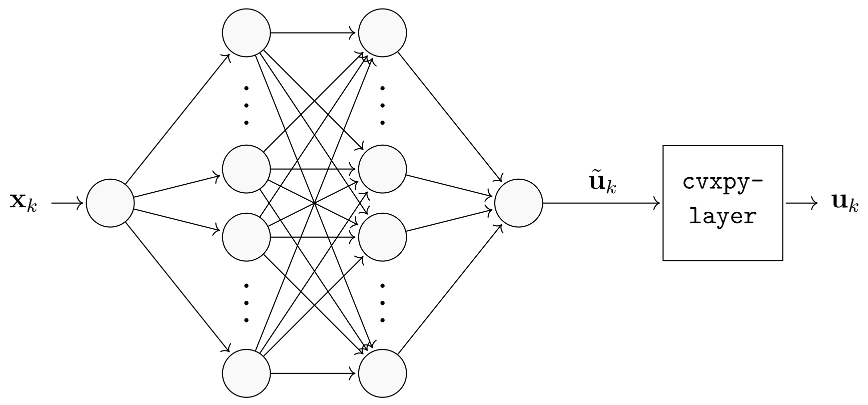


Figure 5.2.1: Schematic of the PNN with the incorporated cxvpy-layer [1].

By incorporating the projection block as its own layer in the network, the network will gain knowledge of the considered MPC problem. The projection block will in this way also influence the parameter updates during the network's learning process and is therefore required to fulfill certain properties.

As described in Chapter 4, the training of the neural network progresses by the use of back-propagation, which makes use of the gradient of the entire network with respect to the network's learnable parameters. It is therefore necessary that the projection block admits a gradient operation, and should thus be a differentiable block. Two frameworks for implementing such a network structure are the OptNet [4] and cvxpylayers [1] libraries in Python. This work employs the cvxpylayers framework, which was introduced in Section 4.4.

The cvxpy-layer realizes the projection by solving a constrained optimization problem of the form in (4.7). Defining the feasible region as

$$\mathcal{R}(\mathbf{x}_k) = \{\mathbf{u} \mid \mathbf{A}\mathbf{x}_k + \mathbf{B}\mathbf{u} \in \mathcal{C}_\infty, \mathbf{u} \in \mathcal{U}\} \tag{5.1}$$

where $\mathcal{U}$ is a set of input constraints, will then ensure the projection comes with feasibility guarantees on all future states and control trajectories. To see this, consider the two polytopic sets

$$\mathcal{C}_\infty = \{\mathbf{x} \in \mathbb{R}^n \mid \mathbf{C}_x\mathbf{x} \leq \mathbf{d}_x\} \tag{5.2}$$
$$\mathcal{U} = \{\mathbf{u} \in \mathbb{R}^m \mid \mathbf{C}_u\mathbf{u} \leq \mathbf{d}_u\}. \tag{5.3}$$

Rewriting (5.1) using (5.2) and (5.3), yields the constraints

$$\begin{aligned} &\mathbf{C}_x\mathbf{B}\mathbf{u}_k \leq \mathbf{d}_x - \mathbf{C}_x\mathbf{A}\mathbf{x}_k \\ &\mathbf{C}_u\mathbf{u} \leq \mathbf{d}_u \end{aligned} \tag{5.4}$$

and the optimization problem becomes

$$\begin{aligned} \underset{\mathbf{u}_k}{\arg\min} \quad & \|\tilde{\mathbf{u}}_k - \mathbf{u}_k\|_2^2 \\ \text{s.t.} \quad & \mathbf{C}_x\mathbf{B}\mathbf{u}_k \leq \mathbf{d}_x - \mathbf{C}_x\mathbf{A}\mathbf{x}_k \\ & \mathbf{C}_u\mathbf{u}_k \leq \mathbf{d}_u \end{aligned} \tag{5.5}$$

where $\tilde{\mathbf{u}}_k$ is the input to the cvxpy-layer, $\mathbf{x}_k$ the input to the network, and $\mathbf{u}_k$ the network output.

5.3 LQR Projection Neural Network

The LQR Projection Neural Network (LQR-PNN) incorporates both the projection layer based neural netwok as well as an LQR structure. The LQR structure is introduced to enforce stability of the network, and to act as a safeguard against the neural network producing unstable solutions. As shown in the schematic in Figure 5.3.1, the sum of the outputs from the LQR block and BBNN block are fed into the cvxpy-layer. This network thus presents a trade-off between the infinite horizon LQR solution and the learnt neural network solution.

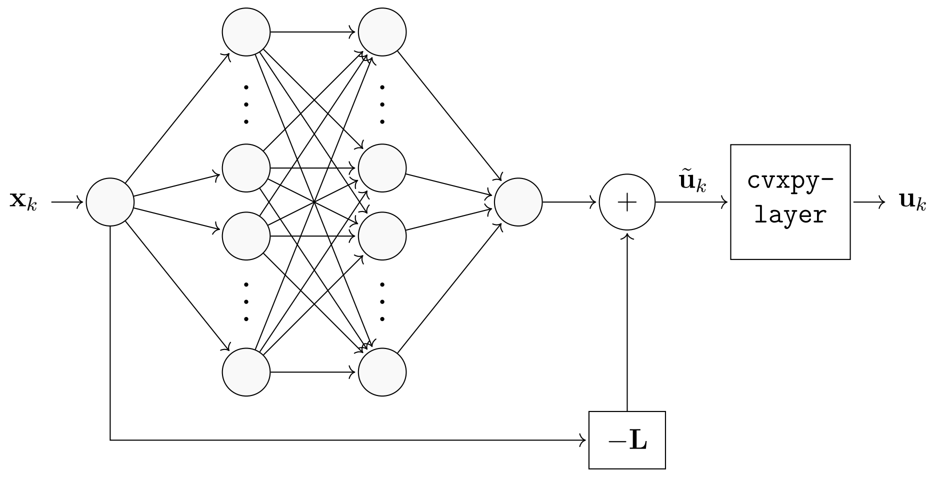


Figure 5.3.1: The structure of the LQR-PNN, where **L** is the feedback gain matrix obtained from solving the related infinite horizon LQR problem.

5.4 LSTM Recurrent Neural Network

A potential drawback of the feed-forward networks is their inability to learn temporal dependencies between data. For this reason, this thesis also investigates a recurrent neural network in the form of an LSTM. This network structure will be referred to as the LSTM-RNN.

The LSTM-RNN consists of one input layer, one hidden LSTM unit with ten neurons each, and one output layer. As in the case of feed-forward networks, the input layer is of size n and the output layer of size m. The difference lies in that the network considers a sequence of states, i.e. a sub-trajectory, of length l at each time instant. The unrolled structure of the network is depicted in Figure 5.4.1.

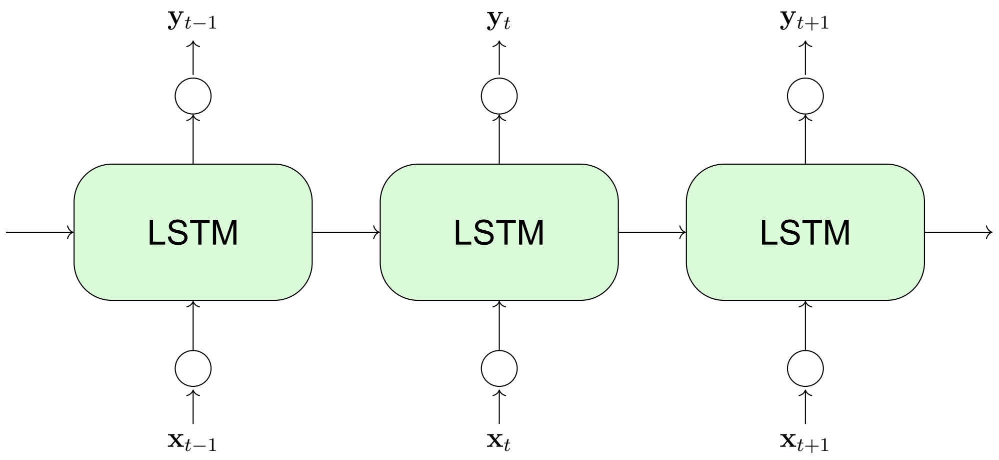


Figure 5.4.1: The unrolled structure of the LSTM-RNN network. The LSTM unit consists of ten neurons. The weights and biases have been omitted for simplicity.

Chapter 6

Training, Simulation, and Evaluation of the Neural Networks

The first section of this chapter presents the proposed method for generating the training and test data. It describes in detail the algorithm for sampling the state space and shows an example of the same. It also discusses how the corresponding control actions are computed.

The following section introduces the training procedure and specifies the choice of loss function and optimizer. It describes in detail also the structure of the training datasets for the two different network families. The last section then defines the two performance metrics used to evaluate the performances of the networks.

6.1 Data Generation

A central aspect of any kind of function approximation technique is the available dataset and in particular its relation to the learnt mapping. To ensure the learning is meaningful, the input space should be sampled such that the resulting approximator generalizes well, i.e., generates accurate predictions even when encountering new, before unseen data. This is especially important for real-life applications such as vehicle control, as one generally cannot expect to have access to an unlimited amount of diverse training data points covering all possible scenarios.

In the case of learning the simple LQR control law in (3.3), this poses no real difficulty. To find the gain matrix $\mathbf{L}$ from training data, a single set of n linearly independent samples of $\mathbf{x}$ and corresponding $\mathbf{u}$ will suffice. However, the complexity increases significantly should one instead consider the explicit MPC. This would require sampling a set of n linearly independent data points in each of the M regions in (3.8), resulting in a minimum of $n \times M$ input-output pairs $(\mathbf{x}, \mathbf{u})$.

Naturally, this raises questions on how the training space should be spanned and sampled so as to obtain as much information as possible while avoiding redundancy. An intuitive and reasonable approach is to uniformly sample the feasible set $\mathcal{C}_\infty$. To this end, this work investigates a Hit-and-Run (HAR) sampler, which performs a random walk across the space.

6.1.1 Hit-And-Run Algorithm

The Hit-and-Run sampler is a Markov chain Monte Carlo method for sampling uniformly from convex shapes [34]. Essentially, starting from any point in the convex set, the method generates a set, $\mathcal{S}$, of points by walking random distances, λ, in randomly generated unit directions. The steps involved in the HAR sampler are detailed in Algorithm 3.

This sampling approach is motivated by the nice convergence properties of the HAR, which ensure that the generated datapoints cover the feasability region in a reasonably uniform manner [34, 45]. This then in turn ensures that the network will have observed training samples that span the entire feasible set on an average, thereby aiding the its ability to generalize.

An example is provided in Figure 6.1.1. The plot shows a set of 1000 points sampled from the region in Figure 2.3.1.

Algorithm 3 Hit-and-Run Sampler

```
 1: procedure HIT-AND-RUN ($\mathcal{C}_\infty, N_S$)
 2:     Pick random point $\mathbf{x} \in \mathcal{C}_\infty = \{\mathbf{x} \in \mathbb{R}^n \mid \mathbf{C}_x\mathbf{x} \leq \mathbf{d}_x\}$
 3:     $\mathcal{S} \leftarrow \{\mathbf{x}\}$
 4:     for $i = 1, \dots, N_S - 1$ do
 5:         $\lambda_i \leftarrow \infty$
 6:         Generate random unit direction $\mathbf{d}_i$
 7:         for $(\mathbf{c}, d)$ in $(\mathbf{C}_x, \mathbf{d}_x)$ do
 8:             $\lambda \leftarrow \dfrac{d - \mathbf{c} \cdot \mathbf{x}}{\mathbf{c} \cdot \mathbf{d}_i}$
 9:             if $\lambda > 0$ then        ▷ To ensure walking in dir. in $\mathbf{d}_i$, not $-\mathbf{d}_i$
10:                 $\lambda_i \leftarrow \min(\lambda_i, \lambda)$
11:             end if
12:         end for
13:         $\lambda_i \leftarrow$ drawn from $\mathbb{U}[0, \lambda_i)$
14:         $\mathbf{x} \leftarrow \mathbf{x} + \lambda_i \mathbf{d}_i$
15:         $\mathcal{S} \leftarrow \mathcal{S} \cup \{\mathbf{x}\}$
16:     end for
17:     return $\mathcal{S}$
18: end procedure
```

6.1.2 Online MPC Simulation

Once the training and test inputs have been sampled from the input space, their corresponding outputs are generated by using OSQP[1] [43] to solve the MPC problem of interest. These sets of pairs then form the training and

[1] An open-source solver for quadratic programming.

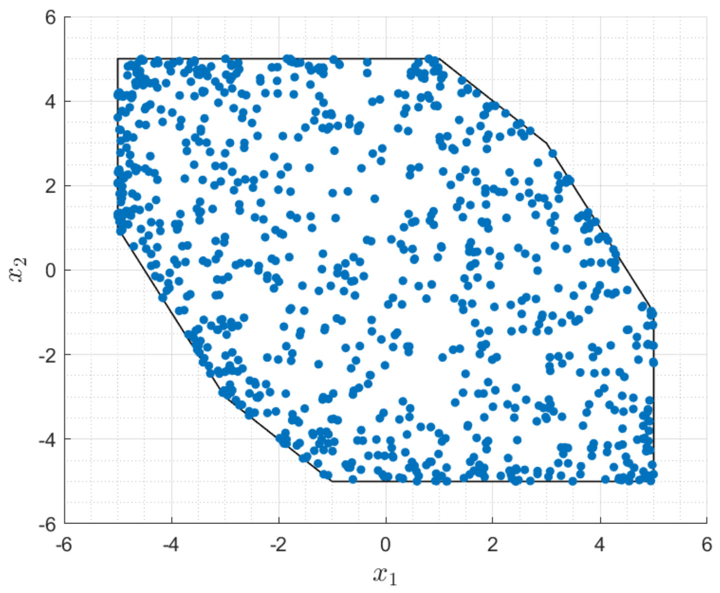


Figure 6.1.1: Hit-and-run sampling of 1000 points from $\mathcal{C}_\infty$ in Figure 2.3.1.

test sets used for training and evaluating the feed-forward networks. With N_t and N_e denoting the number of samples in the two sets, the training and test set can be described in mathematical terms by

$$\mathcal{T} = \{(\mathbf{x}_1^*, \mathbf{u}_1^*), (\mathbf{x}_2^*, \mathbf{u}_2^*), \ldots, (\mathbf{x}_{N_t}^*, \mathbf{u}_{N_t}^*)\} \tag{6.1}$$

and

$$\mathcal{E} = \{(\tilde{\mathbf{x}}_1^*, \tilde{\mathbf{u}}_1^*), (\tilde{\mathbf{x}}_2^*, \tilde{\mathbf{u}}_2^*), \ldots, (\tilde{\mathbf{x}}_{N_e}^*, \tilde{\mathbf{u}}_{N_e}^*)\} \tag{6.2}$$

respectively. The notation $(\tilde{\mathbf{x}}, \tilde{\mathbf{u}})$ is introduced to emphasize that the two sets are generated independently.

Unlike the FFNN's, the recurrent neural network should be trained on sequential data and thus requires the datasets to consist of trajectories of input-out put pairs, rather than independent instances of such. The data generation procedure above is therefore modified, to account for this. The difference lies in the sampled input states being treated as randomly generated initial states, from which full trajectories of length l are generated using OSQP. The training and test set for the RNN will thus be of the form

$$\begin{aligned}\mathcal{T} = \{&\{(\mathbf{x}_1^*, \mathbf{u}_1^*)^{(1)}, (\mathbf{x}_2^*, \mathbf{u}_2^*)^{(1)}, \dots, (\mathbf{x}_l^*, \mathbf{u}_l^*)^{(1)}\},\\ &\{(\mathbf{x}_1^*, \mathbf{u}_1^*)^{(2)}, (\mathbf{x}_2^*, \mathbf{u}_2^*)^{(2)}, \dots, (\mathbf{x}_l^*, \mathbf{u}_l^*)^{(2)}\},\\ &\qquad\vdots\\ &\{(\mathbf{x}_1^*, \mathbf{u}_1^*)^{(N_t)}, (\mathbf{x}_2^*, \mathbf{u}_2^*)^{(N_t)}, \dots, (\mathbf{x}_l^*, \mathbf{u}_l^*)^{(N_t)}\}\}\end{aligned} \tag{6.3}$$

and

$$\begin{aligned}\mathcal{E} = \{&\{(\tilde{\mathbf{x}}_1^*, \tilde{\mathbf{u}}_1^*)^{(1)}, (\tilde{\mathbf{x}}_2^*, \tilde{\mathbf{u}}_2^*)^{(1)}, \dots, (\tilde{\mathbf{x}}_l^*, \tilde{\mathbf{u}}_l^*)^{(1)}\},\\ &\{(\tilde{\mathbf{x}}_1^*, \tilde{\mathbf{u}}_1^*)^{(2)}, (\tilde{\mathbf{x}}_2^*, \tilde{\mathbf{u}}_2^*)^{(2)}, \dots, (\tilde{\mathbf{x}}_l^*, \tilde{\mathbf{u}}_l^*)^{(2)}\},\\ &\qquad\vdots\\ &\{(\tilde{\mathbf{x}}_1^*, \tilde{\mathbf{u}}_1^*)^{(N_e)}, (\tilde{\mathbf{x}}_2^*, \tilde{\mathbf{u}}_2^*)^{(N_e)}, \dots, (\tilde{\mathbf{x}}_l^*, \tilde{\mathbf{u}}_l^*)^{(N_e)}\}\}\end{aligned} \tag{6.4}$$

respectively, where the superscript (i) denotes the index of the trajectory, i.e., the index of the initial state from the input samples.

6.2 Training

Following the generation of datasets, the networks are trained using a supervised learning method. During training, the network's learnable parameters, θ, are updated by minimizing the Mean-Squared Error (MSE) loss function

$$L(\theta) = \frac{1}{N_t} \sum_{i=0}^{N_t-1} (\mu(\mathbf{x}_i, \theta) - \mathbf{u}_i^*)^2, \tag{6.5}$$

where $\mu(\mathbf{x}_i, \theta)$ is the output of the network, and $\mathbf{u}_i^*$ the target value.

In order to increase the training speed, the network is trained on mini-batches of data rather than on the whole dataset at once. In this way, the MSE-loss (6.5) will be computed following each batch, which in turn results in the parameters being updated after each batch. The mini-batches are obtained by splitting the training set into smaller subsets as follows

$$\begin{aligned}
\mathcal{B}_1 =& \{\mathcal{T}(1), \mathcal{T}(2), \ldots, \mathcal{T}(b)\}, \\
\mathcal{B}_2 =& \{\mathcal{T}(b+1), \mathcal{T}(b+2), \ldots, \mathcal{T}(2b)\}, \\
&\vdots \\
\mathcal{B}_{N_t/s_B} =& \{\mathcal{T}(N_t - s_B + 1), \mathcal{T}(N_t - s_B + 2), \ldots, \mathcal{T}(N_t)\},
\end{aligned} \tag{6.6}$$

where $\mathcal{T}(i)$, with a slight abuse of notation, denotes the i:th element of the training set, and s_B the batch size.

The gradient descent-based *Adam* opitmizer [25] is used to backpropagate the loss and update the parameters θ following each batch. Once all the mini-batches have been iterated over, one training epoch is completed. The networks are trained for as many epochs required to reach convergence.

6.3 Performance Metrics

The feed-forward network architectures are evaluated in terms of the two performance metrics:

1. The Normalized Mean Square Error (NMSE) in dB, which is defined as

$$\text{NMSE} = 10 \log_{10} \left(\frac{\|\mathbf{u} - \mathbf{u}^*\|_2^2}{\|\mathbf{u}^*\|_2^2} \right) \tag{6.7}$$

   where $\mathbf{u}$ denotes the output of the network, and $\mathbf{u}^*$ the desired output, and

2. The normalized control cost J_n, defined as the control cost, J, in Equation (3.7), normalized by $\mathbf{x}_0^\top \mathbf{x}_0$:

$$J_n = \frac{\mathbf{x}_N^\top \mathbf{Q}_N \mathbf{x}_N + \sum_{k=0}^{N-1} \left[\mathbf{x}_k^\top \mathbf{Q} \mathbf{x}_k + \mathbf{u}_k^\top \mathbf{R} \mathbf{u}_k \right]}{\mathbf{x}_0^\top \mathbf{x}_0} \tag{6.8}$$

where $\mathbf{x}_0$ is the initial state of the trajectory.

Both metrics are evaluated on test data, previously unseen by the networks during training. The NMSE helps evaluate the control law predicted by the network with respect to the ground truth, whereas the control cost measures how well the control law is in terms of minimizing the control objective: the smaller the J_n, the better the control achieved.

Chapter 7

Results of Numerical Examples

The three feed-forward networks BBNN, PNN, and LQR-PNN have been trained and evaluated on two numerical examples: a two-dimensional system and a four-dimensional system. They were evaluated in terms of the NMSE metric in (6.7) and the control cost metric in (6.8). The LSTM-RNN was implemented only for a family of two-dimensional systems.

For the control cost evaluation, a separate set of initial states was sampled from $\mathcal{C}_\infty$ using the HAR in Algorithm 3. Starting from these, the trained networks were simulated in closed loop for ten time steps to generate trajectories. These were then compared to the trajectories generated by the OSQP (online MPC).

7.1 2D-example

The first example considers the following two-dimensional system from [9]

$$\mathbf{x}_{k+1} = \mathbf{A}\mathbf{x}_k + \mathbf{B}\mathbf{u}_k = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \mathbf{x}_k + \begin{bmatrix} 0 \\ 1 \end{bmatrix} u_k \tag{7.1}$$

subject to the constraints

$$\begin{bmatrix} -5 \\ -5 \end{bmatrix} \leq \mathbf{x}_k \leq \begin{bmatrix} 5 \\ 5 \end{bmatrix}, \quad -1 \leq u_k \leq 1 \tag{7.2}$$

and with cost parameters

$$\mathbf{Q}_N = \mathbf{Q} = \mathbf{I}_2 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \quad \mathbf{R} = 10, \quad N = 3. \tag{7.3}$$

It holds from the Hautus lemmas in (2.3) and (2.4) that the system is stabilizable and that the pair $(\mathbf{Q}^{1/2}, \mathbf{A})$ is detectable. The maximal control invariant set was computed using Algorithm 1 and the region is plotted in Figure 7.1.1.

The MPC problem for the example is then given by

$$\begin{aligned} \underset{\mathbf{u}_{0:2}}{\arg\min} \quad & J(\mathbf{x}_0) = \mathbf{x}_2^\top \mathbf{I}_2 \mathbf{x}_2 + \sum_{k=0}^{2} (\mathbf{x}_k^\top \mathbf{I}_2 \mathbf{x}_k + 10u_k^2) \\ \text{s.t.} \quad & (7.1) \text{ and } (7.2),\ k = 0, 1, 2 \end{aligned} \tag{7.4}$$

and the related LQR problem as

$$\underset{\mathbf{u}_{0:\infty}}{\arg\min} \quad J_\infty = \sum_{k=0}^{\infty} (\mathbf{x}_k^\top \mathbf{I}_2 \mathbf{x}_k + 10u_k^2). \tag{7.5}$$

The LQR was solved so as to obtain the gain matrix $\mathbf{L}$ for the LQR block in the LQR-PNN. Using (3.3) and (3.4), it was evaluated to

$$\mathbf{L} = \begin{bmatrix} 0.20540 & 0.78352 \end{bmatrix}. \tag{7.6}$$

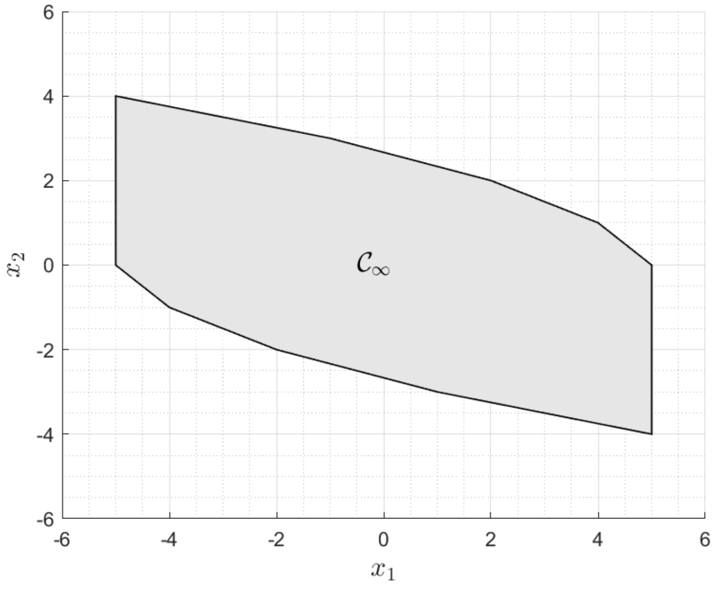
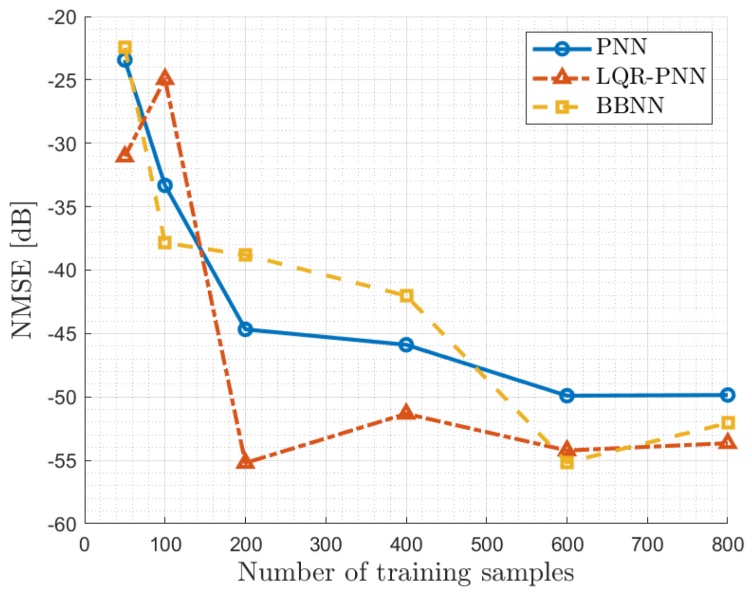
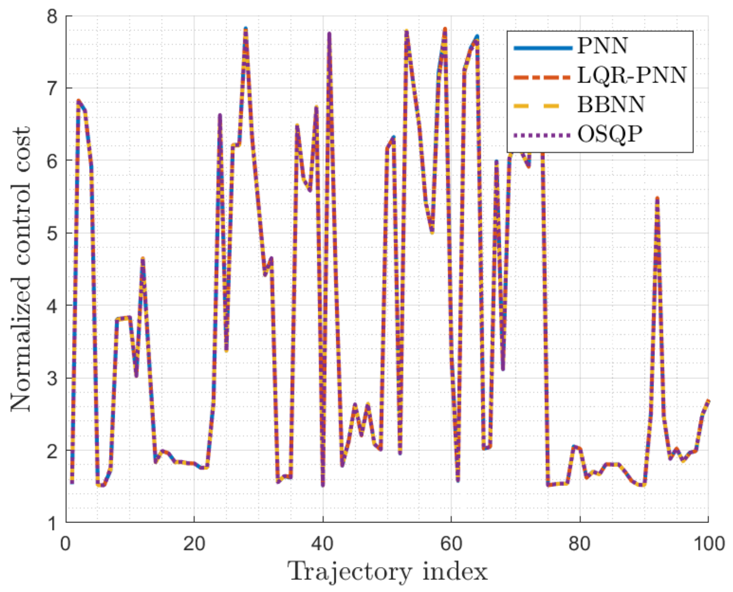


Figure 7.1.1: The maximal control invariant set, $\mathcal{C}_\infty$, for system (7.1) subject to constraints (7.2).

Figure 7.1.2a shows the NMSE as a function of the size of the training dataset. Figure 7.1.2b shows the normalized control cost, J_n, computed for 100 trajectories for the different control laws. For the NMSE evaluation, a test set of 500 samples was used. For the control cost evaluation, the networks were trained on a dataset of 1000 samples.



(a) Comparison of the computed NMSE on test data for the different control laws for the two-dimensional example.



(b) Comparison of the control costs computed for the different control laws for 100 test trajectories for the two-dimensional example.

Figure 7.1.2: Evaluation of the computed control laws for the double integrator example.

7.2 4D-Example

The second example considers the 4D system from [10]

$$\mathbf{x}_{k+1} = \mathbf{A}\mathbf{x}_k + \mathbf{B}\mathbf{u}_k = \begin{bmatrix} 0.7 & -0.1 & 0.0 & 0.0 \\ 0.2 & -0.5 & 0.1 & 0.0 \\ 0.0 & 0.1 & 0.1 & 0.0 \\ 0.5 & 0.0 & 0.5 & 0.5 \end{bmatrix} \mathbf{x}_k + \begin{bmatrix} 0.0 & 0.1 \\ 0.1 & 1.0 \\ 0.1 & 0.0 \\ 0.0 & 0.0 \end{bmatrix} \mathbf{u}_k \tag{7.7}$$

subject to the constraints

$$\begin{bmatrix} -6.0 \\ -6.0 \\ -1.0 \\ -0.5 \end{bmatrix} \leq \mathbf{x}_k \leq \begin{bmatrix} 6.0 \\ 6.0 \\ 1.0 \\ 0.5 \end{bmatrix}, \quad \begin{bmatrix} -5.0 \\ -5.0 \end{bmatrix} \leq \mathbf{u}_k \leq \begin{bmatrix} 5.0 \\ 5.0 \end{bmatrix}, \tag{7.8}$$

and with cost parameters

$$\mathbf{Q}_N = \mathbf{Q} = \begin{bmatrix} 1.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix}, \quad \mathbf{R} = \begin{bmatrix} 1.0 & 0.0 \\ 0.0 & 1.0 \end{bmatrix}, \quad N = 10. \tag{7.9}$$

As in the previous example, it holds from the Hautus lemmas in (2.3) and (2.4) that this system is stabilizable and that the pair $(\mathbf{Q}^{1/2}, \mathbf{A})$ is detectable. Algorithm 1 was again used to compute $\mathcal{C}_\infty$.

The MPC problem is given by

$$\begin{aligned} \underset{\mathbf{u}_{0:2}}{\arg\min} \quad & J(\mathbf{x}_0) = \mathbf{x}_2^\top \mathbf{I}_4 \mathbf{x}_2 + \sum_{k=0}^{2} (\mathbf{x}_k^\top \mathbf{I}_4 \mathbf{x}_k + \mathbf{u}_k^\top \mathbf{I}_2 \mathbf{u}_k) \\ \text{s.t.} \quad & (7.7) \text{ and } (7.8),\ k = 0, 1, 2 \end{aligned} \tag{7.10}$$

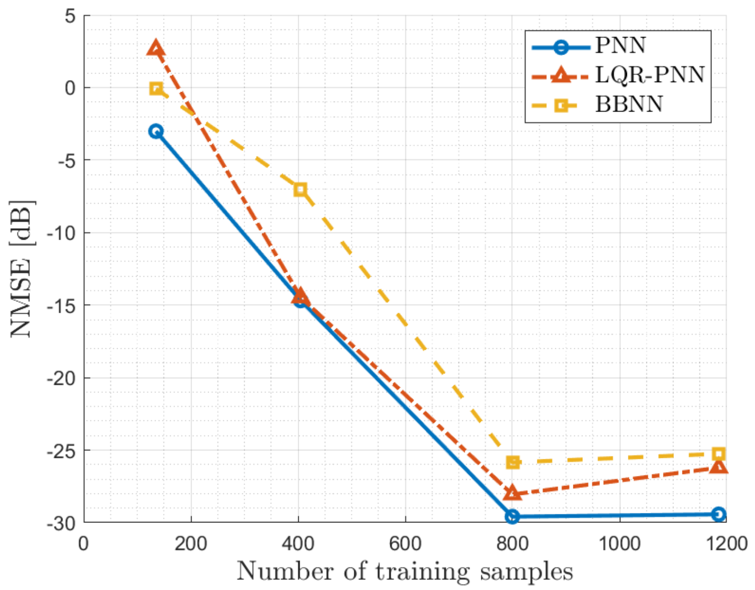
and the corresponding LQR problem by

$$\underset{\mathbf{u}_{0:\infty}}{\arg\min} \quad J_\infty = \sum_{k=0}^{\infty} (\mathbf{x}_k^\top \mathbf{I}_4 \mathbf{x}_k + \mathbf{u}_k^\top \mathbf{I}_2 \mathbf{u}_k). \tag{7.11}$$

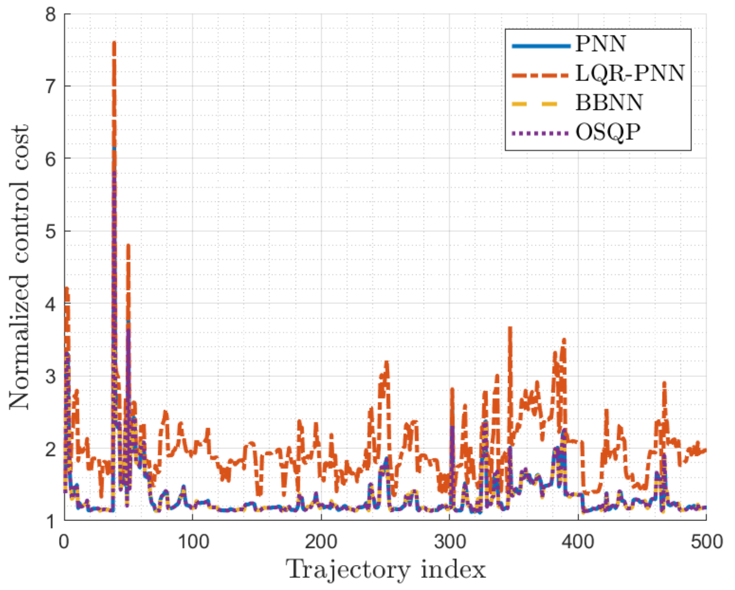
The gain matrix is evaluated to

$$\mathbf{L} = \begin{bmatrix} 0.0471 & 0.015123 & 0.033994 & 0.015731 \\ 0.16897 & -0.26906 & 0.064319 & 0.010699 \end{bmatrix}. \tag{7.12}$$

The plot in Figure 7.2.1a shows the NMSE as a function of the size of the training dataset. Figure 7.2.1b shows the normalized control cost, J_n, computed for 500 trajectories for the different control laws. Note especially how the trajectories generated by the LQR-PNN consistently admit the highest cost. For the NMSE evaluation, a test dataset of 500 samples was used. For the control cost evaluation, the networks were trained on a set of 7000 samples.



(a) Comparison of the computed NMSE on test data for the different control laws for the 4D-example.



(b) Comparison of the control costs computed for the different control laws for 100 test trajectories for the 4D-example.

Figure 7.2.1: Evaluation of the computed control laws for the 4D-example.

7.3 2D-examples for the LSTM-RNN

The LSTM-RNN network was trained on four different MPC problems, examples A-D below, and then tested independently for robustness on two previously unseen examples, examples E-F below. The training set consisted of 1200 trajectories, 300 from each example. The two separate test sets consisted of 300 trajectories each. The initial states of all trajectories were sampled from the $\mathcal{C}_\infty$ of the individual systems.

All systems are subject to the constraints in (7.2), and the cost parameters of the corresponding MPC problems are defined as in (7.3). The maximal control invariant sets for the different systems subject to (7.2) were computed using Algorithm 1 and are shown in Figures 7.3.1 and 7.3.2.

To evaluate the performance of the network, the NMSE was computed for the two test cases. The results are presented in Table 7.3.1.

Example A. The system is defined as

$$\mathbf{x}_{k+1} = \mathbf{A}\mathbf{x}_k + \mathbf{B}\mathbf{u}_k = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \mathbf{x}_k + \begin{bmatrix} 0 \\ 1 \end{bmatrix} u_k.$$

Example B. The system is defined as

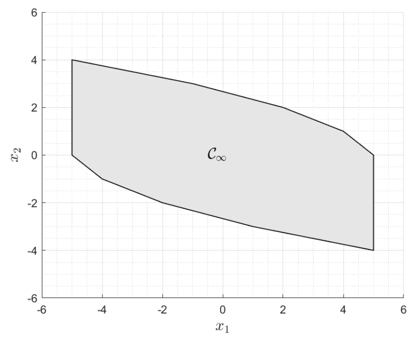
$$\mathbf{x}_{k+1} = \mathbf{A}\mathbf{x}_k + \mathbf{B}\mathbf{u}_k = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \mathbf{x}_k + \begin{bmatrix} 1 \\ 1 \end{bmatrix} u_k.$$

Example C. The system is defined as

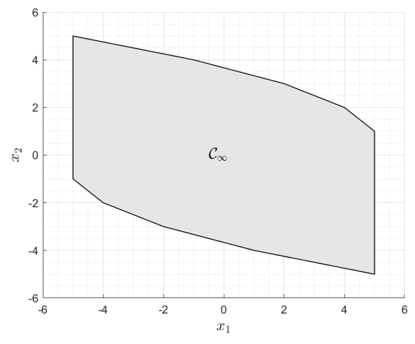
$$\mathbf{x}_{k+1} = \mathbf{A}\mathbf{x}_k + \mathbf{B}\mathbf{u}_k = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \mathbf{x}_k + \begin{bmatrix} 2 \\ 1 \end{bmatrix} u_k.$$

Example D. The system is defined as

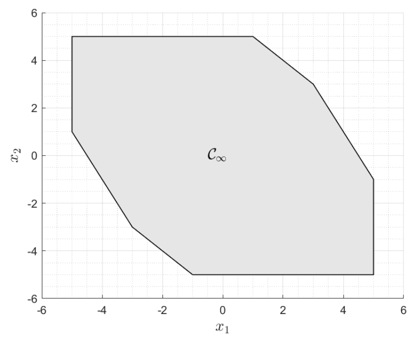
$$\mathbf{x}_{k+1} = \mathbf{A}\mathbf{x}_k + \mathbf{B}\mathbf{u}_k = \begin{bmatrix} 2 & 0 \\ 2 & 1 \end{bmatrix} \mathbf{x}_k + \begin{bmatrix} 1 \\ 0 \end{bmatrix} u_k.$$



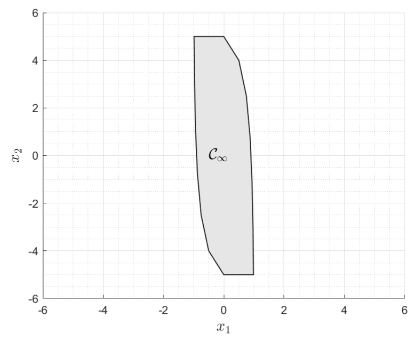
(a) $\mathcal{C}_\infty$ for Example A.



(b) $\mathcal{C}_\infty$ for Example B.



(c) $\mathcal{C}_\infty$ for Example C.



(d) $\mathcal{C}_\infty$ for Example D.

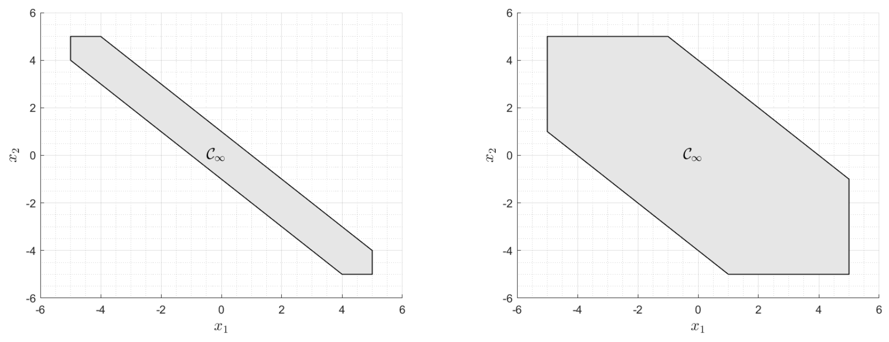
Figure 7.3.1: $\mathcal{C}_\infty$ for the examples the LSTM-RNN network was trained on.

Example E. The system is defined as

$$\mathbf{x}_{k+1} = \mathbf{A}\mathbf{x}_k + \mathbf{B}\mathbf{u}_k = \begin{bmatrix} 1 & 1 \\ 2 & 2 \end{bmatrix} \mathbf{x}_k + \begin{bmatrix} 1 \\ 1 \end{bmatrix} u_k.$$

Example F. The system is defined as

$$\mathbf{x}_{k+1} = \mathbf{A}\mathbf{x}_k + \mathbf{B}\mathbf{u}_k = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \mathbf{x}_k + \begin{bmatrix} 3 \\ 1 \end{bmatrix} u_k.$$



(a) $\mathcal{C}_\infty$ for Example E. (b) $\mathcal{C}_\infty$ for Example F.

Figure 7.3.2: $\mathcal{C}_\infty$ for the examples the LSTM-RNN network was tested on.

Table 7.3.1: The computed NMSE from testing the LSTM-RNN on the Examples E and F.

Example	**NMSE**	**NMSE [dB]**
E	0.1233	-9.0889
F	0.0672	-11.7284

Chapter 8

Conclusions

A framework for offline training and evaluation of neural networks approaches for MPC has been presented. The basic idea is to approximate the MPC mapping from state to control input with constrained ReLU based neural networks including a projection layer. Recent papers, [8, 10], have motivated this structure from eMPC, and that a continuous PWA function on polyhedra can be represented exactly by a deep enough ReLU neural network. The role of the projection layer is to guarantee recursive feasibility of the trajectories. The main aspects that have been studied are:

- *Network implementation:* The Python libraries PyTorch [37] and cvxpylayers [1] have been used to implement this framework. The solver OSQP [43] have been used to generate training data and for evaluating the resulting controllers. This tool box showed to provide efficient means for training and evaluation.

- *State space sampling:* A key factor is the generation of samples for the training of the networks. This is a challenge when the dimension of the state space increase. This work proposed a HAR algorithm for sampling the state space.

- *Evaluation of network structures:* Evaluation of the resulting controller based on trajectories and normalized cost-functions. The numerical tests showed the trade-off between the number of training data and the performance of the resulting controller.

This work is a first step to more systematic methods for training and evaluation of neural networks implemented MPC.

8.1 Network Structures

The results from the feed-forward network evaluations on the four dimensional system showed weak trends of increased performance for the network structures incorporating the cvxpy-layer. For the two dimensional system, there were no apparent differences in terms of performance between the control laws. This is likely due to the lower dimension of the problem, which makes it easier for the networks to learn the underlying structure. The LSTM-RNN was only tested for robustness and the results suggests that the network generalizes to new MPC problems.

It is important to note that before any general conclusions regarding the relation between performance and network structure can be made, further studies including more extensive experiments are necessary. In particular, no effort has been made in this work to optimize either network structure in terms of the total number of neurons or hidden layers. Also, because of time limitations, the FFNN's and the LSTM-RNN were not properly compared and therefore nothing can be said in regards to the difference in performance between them.

8.2 Future Work

Since the long-term goal is to implement network based controllers in real-life applications, a next natural step would be to train and test the network controllers for reference tracking. One idea is to train the networks for tracking sinusoidal references and then evaluate them on arbitrary periodic signals, which can be represented as a sum of sinusoids. An intuitive approach would be to extend the networks to also consider both the current as well as the past and future reference signals as inputs. It would then make sense to compare the performances between the feed-forward structures and the recurrent one.

A major restriction of current methods is that they train for specific models. Another interesting idea for future work would thus be to use model parameters as well as states as inputs to the constrained network. This would increase the complexity and the need for even more structured training data generation, but could potentially result in more versatile network controllers. In addition, the networks should be trained and tested on higher-dimensional systems.

Another aspect, which has not been considered here, is the computational cost. As mentioned in the introduction, the eMPC was mainly introduced as a means for decreasing the computation required online. It is therefore reasonable to assume that it would be of interest to also consider the online time complexity in the comparison of the network controllers.

Bibliography

[1] Agrawal, A., Amos, B., Barratt, S., Boyd, S., Diamond, S., and Kolter, Z. “Differentiable Convex Optimization Layers”. In: *Advances in Neural Information Processing Systems*. 2019.

[2] Åkesson, B. and Toivonen, H. “A neural network model predictive controller”. In: *Journal of Process Control* 16 (Oct. 2006), pp. 937–946. DOI: `10.1016/j.jprocont.2006.06.001`.

[3] Alessio, A. and Bemporad, A. “A Survey on Explicit Model Predictive Control”. In: *Nonlinear Model Predictive Control: Towards New Challenging Applications*. Ed. by L. Magni, D. M. Raimondo, and F. Allgöwer. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 345–369. ISBN: 978-3-642-01094-1. DOI: `10.1007/978-3-642-01094-1_29`. URL: `https://doi.org/10.1007/978-3-642-01094-1_29`.

[4] Amos, B. and Kolter, J. Z. “OptNet: Differentiable Optimization as a Layer in Neural Networks”. In: *ICML*. 2017.

[5] Åström, K. and Murray, R. “Feedback Systems: An Introduction for Scientists and Engineers”. In: *Feedback Systems: An Introduction for Scientists and Engineers* (Jan. 2008).

[6] Bemporad, A. “Explicit Model Predictive Control”. In: *Encyclopedia of Systems and Control*. Ed. by J. Baillieul and T. Samad. London: Springer London, 2013, pp. 1–9. ISBN: 978-1-4471-5102-9. DOI: `10 .`

1007/978-1-4471-5102-9_10-1. URL: https://doi.org/10.1007/978-1-4471-5102-9_10-1.

[7] Bishop, C. M. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006. ISBN: 0387310738.

[8] Blanchini, F. "Set invariance in control". In: *Automatica* 35.11 (1999), pp. 1747–1767. ISSN: 0005-1098. DOI: https://doi.org/10.1016/S0005-1098(99)00113-2. URL: http://www.sciencedirect.com/science/article/pii/S0005109899001132.

[9] Borrelli, F., Bemporad, A., and Morari, M. *Predictive Control for Linear and Hybrid Systems*. 1st. USA: Cambridge University Press, 2017. ISBN: 1107652871.

[10] Chen, S., Saulnier, K., Atanasov, N., Lee, D. D., Kumar, V., Pappas, G. J., and Morari, M. "Approximating Explicit Model Predictive Control Using Constrained Neural Networks". In: *2018 Annual American Control Conference (ACC)*. 2018, pp. 1520–1527.

[11] Chen, S. W., Wang, T., Atanasov, N., Kumar, V., and Morari, M. *Large Scale Model Predictive Control with Neural Networks and Primal Active Sets*. 2019. arXiv: 1910.10835 [cs.LG].

[12] Cimini, G., Bernardini, D., Levijoki, S., and Bemporad, A. "Embedded Model Predictive Control With Certified Real-Time Optimization for Synchronous Motors". In: *IEEE Transactions on Control Systems Technology* (2020), pp. 1–8.

[13] Dangeti, P. *Statistics for Machine Learning: Techniques for Exploring Supervised, Unsupervised, and Reinforcement Learning Models with Python and R*. Packt Publishing, 2017. ISBN: 1788295757.

[14] Di, W., Bhardwaj, A., and Wei, J. *Deep Learning Essentials: Your Hands-on Guide to the Fundamentals of Deep Learning and Neural Network Modeling*. Packt Publishing, 2018. ISBN: 1785880365.

[15] Diamond, S. and Boyd, S. “CVXPY: A Python-Embedded Modeling Language for Convex Optimization”. In: *Journal of Machine Learning Research* 17.83 (2016), pp. 1–5.

[16] Glad, T. and Ljung, L. *Control Theory: Multivariable and Nonlinear Methods*. 1st. 11 New Fetter Lane, London EC4P 4EE: Taylor & Francis, 2000. ISBN: 0748408789.

[17] Goodwin, G., Seron, M. M., and Don, J. A. de. *Constrained Control and Estimation: An Optimisation Approach*. 1st. Springer Publishing Company, Incorporated, 2010. ISBN: 1849968837.

[18] Grant, M., Boyd, S., and Ye, Y. “Disciplined convex programming”. In: *Global Optimization: From Theory to Implementation, Nonconvex Optimization and Its Application Series*. Springer, 2006, pp. 155–210.

[19] Graves, A. *Supervised Sequence Labelling with Recurrent Neural Networks*. Studies in Computational Intelligence. Berlin: Springer, 2012. DOI: `10.1007/978-3-642-24797-2`.

[20] Grieder, P., Borrelli, F., Torrisi, F., and Morari, M. “Computation of the constrained infinite time linear quadratic regulator”. In: *Proceedings of the 2003 American Control Conference, 2003*. Vol. 6. 2003, 4711–4716 vol.6.

[21] Haykin, S. *Neural Networks: A Comprehensive Foundation*. 1st. USA: Prentice Hall PTR, 1994. ISBN: 0023527617.

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

[23] Hochreiter, S. and Schmidhuber, J. “Long Short-term Memory”. In: *Neural computation* 9 (Dec. 1997), pp. 1735–80. DOI: `10.1162/neco.1997.9.8.1735`.

[24] Kerrigan, E. “Robust Constraint Satisfaction: Invariant Sets and Predictive Control”. PhD thesis. Department of Engineering, University of Cambridge, Cambridge, 2000.

[25] Kingma, D. P. and Ba, J. “Adam: A Method for Stochastic Optimization”. In: *CoRR* abs/1412.6980 (2014).

[26] Kumar, S., Tulsyan, A., Gopaluni, B., and Loewen, P. “A Deep Learning Architecture for Predictive Control”. In: *IFAC-PapersOnLine* 51 (Jan. 2018), pp. 512–517. DOI: `10.1016/j.ifacol.2018.09.373`.

[27] Kvasnica, M., Holaza, J., Takacs, B., and Ingole, D. “Design and Verification of Low-Complexity Explicit MPC Controllers in MPT3 (Extended version)”. In: July 2015. DOI: `10.1109/ECC.2015.7330929`.

[28] Kwon, W. and Han, S. *Receding Horizon Control: Model Predictive Control for State Models*. Advanced Textbooks in Control and Signal Processing. Springer London, 2006. ISBN: 9781846280177.

[29] Lanzetti, N., Lian, Y. Z., Cortinovis, A., Dominguez, L., Mercangöz, M., and Jones, C. “Recurrent Neural Network based MPC for Process Industries”. In: *2019 18th European Control Conference (ECC)*. 2019, pp. 1005–1010.

[30] LeCun, Y., Bengio, Y., and Hinton, G. “Deep Learning”. In: *Nature* 521 (May 2015), pp. 436–44. DOI: `10.1038/nature14539`.

[31] Maddalena, E. T., S. Moraes, C. G. da, Waltrich, G., and Jones, C. N. *A Neural Network Architecture to Learn Explicit MPC Controllers from Data*. 2019. arXiv: `1911.10789 [eess.SY]`.

[32] Mandic, D. P. and Chambers, J. *Recurrent Neural Networks for Prediction: Learning Algorithms,Architectures and Stability*. USA: John Wiley Sons, Inc., 2001. ISBN: 0471495174.

[33] McCulloch, W. and Pitts, W. “A Logical Calculus of Ideas Immanent in Nervous Activity”. In: *The Bulletin of Mathematical Biophysics* 5 (1943), pp. 115–133. URL: `https://doi.org/10.1007/BF02478259`.

[34] Mete, H. O. and Zabinsky, Z. B. “Pattern Hit-and-Run for sampling efficiently on polytopes”. In: *Operations Research Letters* 40.1 (2012), pp. 6–11. ISSN: 0167-6377. DOI: `https://doi.org/10.1016/j.orl.2011.11.002`. URL: `http://www.sciencedirect.com/science/article/pii/S0167637711001271`.

[35] Nielsen, M. *Neural Networks and Deep Learning*. Determination Press, 2015. URL: `https://books.google.se/books?id=STDBswEACAAJ`.

[36] Parisini, T. and Zoppoli, R. “A receding-horizon regulator for nonlinear systems and a neural approximation”. In: *Autom.* 31 (1995), pp. 1443–1451.

[37] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. “PyTorch: An Imperative Style, High-Performance Deep Learning Library”. In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett. Curran Associates, Inc., 2019, pp. 8024–8035. URL: `http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf`.

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

[39] Rojas, R. *Neural Networks: A Systematic Introduction*. Berlin, Heidelberg: Springer-Verlag, 1996. ISBN: 3540605053.

[40] Scokaert, P. O. M. and Rawlings, J. B. “Constrained linear quadratic regulation”. In: *IEEE Transactions on Automatic Control* 43.8 (1998), pp. 1163–1169.

[41] Sontag, E. *Mathematical Control Theory: Deterministic Finite-Dimensional Systems*. Jan. 1998. DOI: `10.1007/978-1-4612-0577-7`.

[42] Stathopoulos, G., Korda, M., and Jones, C. N. “Solving the Infinite-Horizon Constrained LQR Problem Using Accelerated Dual Proximal Methods”. In: *IEEE Transactions on Automatic Control* 62.4 (2017), pp. 1752–1767.

[43] Stellato, B., Banjac, G., Goulart, P., Bemporad, A., and Boyd, S. “OSQP: an operator splitting solver for quadratic programs”. In: *Mathematical Programming Computation* (2020), pp. 1867–2957. DOI: `https://doi.org/10.1007/s12532-020-00179-2`.

[44] Winqvist, R., Venkitaraman, A., and Wahlberg, B. “On Training and Evaluation of Neural Network Approaches for Model Predictive Control”. In: *ArXiv* abs/2005.04112 (2020).

[45] Zabinsky, Z. B. and Smith, R. L. “Hit-and-Run Methods”. In: *Encyclopedia of Operations Research and Management Science*. Ed. by S. I. Gass and M. C. Fu. Boston, MA: Springer US, 2013, pp. 721–729. ISBN: 978-1-4419-1153-7. DOI: `10.1007/978-1-4419-1153-7_1145`. URL: `https://doi.org/10.1007/978-1-4419-1153-7_1145`.

[46] Zaccone, G. and Karim, M. R. *Deep Learning with TensorFlow - Second Edition: Explore Neural Networks and Build Intelligent Systems with Python*. 2nd. Packt Publishing, 2018. ISBN: 1788831101.