Nonlinear Approximative Explicit Model Predictive Control Through Neural Networks
Characterizing Architectures and Training Behavior

TOBIAS BOLIN
Nonlinear Approximative Explicit Model Predictive Control Through Neural Networks

Characterizing Architectures and Training Behavior

TOBIAS BOLIN, TBOLIN@KTH.SE
Abstract

Model predictive control (MPC) is a paradigm within automatic control notable for its ability to handle constraints. This ability come at the cost of high computational demand, which until recently has limited use of MPC to slow systems. Recent advances have however enabled MPC to be used in embedded applications, where its ability to handle constraints can be leveraged to reduce wear, increase efficiency and improve overall performance in everything from cars to wind turbines. MPC controllers can be made even faster by precomputing the resulting policy and storing it in a lookup table. A method known as explicit MPC.

An alternative way of leveraging precomputation is to train a neural network to approximate the policy. This is an attractive proposal both due to neural networks ability to imitate policies for nonlinear systems, and results that indicate that neural networks can efficiently represent explicit MPC policies. Limited work has been done in this area. How the networks are setup and trained therefore tends to reflect recent trends in other application areas rather than being based on what is known to work well for approximating MPC policies. This thesis attempts to alleviate this situation by evaluating how some common neural network architectures and training methods performs when used for this purpose. The evaluations are carried out through a literature study and by training several networks with different architectures to replicate the policy of a nonlinear MPC controller tasked with stabilizing an inverted pendulum.

The results suggest that ReLU activation functions give better performance than hyperbolic tangent and SELU functions; and that dropout and batch normalization degrades the ability to approximate policies; and that depth significantly increases the performance. However, the neural network controllers do occasionally exhibit problematic behaviors, such as steady state errors and oscillating control signals close to constraints.
Sammanfattning


Resultaten tyder på att nätverk med ReLU-aktivering ger bättre prestanda än motsvarande nätverk som använder SELU eller tangens hyperbolicus som aktiveringsfunktion. Resultaten visar också att batch normalization och dropout försämrrar nätverkens förmåga att lära sig policyn och att prestandan blir bättre om antalet lager i nätverket ökar. De neurala nätverken uppvisar dock i vissa fall kvalitativa problem, så som statiska fel och oscillerande kontrollsignaler nära begränsningar.
Acknowledgments

I would like to thank my supervisor Patric Jensfelt for all the help with planning and structuring the process of writing this thesis. I would also like to thank Ludvig Ericson for guiding me towards the thesis subject. The support from the other members of the supervision group, Jiongrui Hu, Viktor Tuul and Fanny Radesjö were also deeply appreciated. I have to mention John D. Clark who, despite being very dead for the better part of three decades, reminded me of how empirical science is done through repeated and well documented failures and to be ever grateful for the fact that neural networks are neither corrosive, flammable or high explosive. Finally, I am extremely grateful to my rubber ducks for all the emotional support they have provided during the writing process and for acting as excellent models for the cover.

\[1\text{through repeated and well documented failures}\]
## Experiments and Results

5 Experiments and Results 32

5.1 Experimental setup .............................................. 32
  5.1.1 Data generation .............................................. 33
  5.1.2 Training .................................................... 35

5.2 Divided training ................................................. 35
  5.2.1 Results .................................................... 36
  5.2.2 Discussion ................................................. 37

5.3 Complexity of controllers with and without constraints ... 38
  5.3.1 Result .................................................... 38
  5.3.2 Discussion ................................................ 40

5.4 Comparison of output clamping and activation functions ... 40
  5.4.1 Results .................................................... 41
  5.4.2 Discussion ................................................ 43

5.5 Direct performance evaluation ................................ 46
  5.5.1 Results .................................................... 47
  5.5.2 Discussion ................................................ 50

5.6 Effect of depth on performance ................................ 51
  5.6.1 Results .................................................... 52
  5.6.2 Discussion ................................................ 53

5.7 Batch normalization and Dropout ................................ 54
  5.7.1 Results .................................................... 54
  5.7.2 Discussion ................................................ 55

5.8 Qualitative behavior ............................................. 55
  5.8.1 Results .................................................... 56
  5.8.2 Discussion ................................................ 56

## Conclusions and Further Work

6 Conclusions and Further Work 62

6.1 Conclusions ..................................................... 62

6.2 Societal Impact, Ethics and Sustainability .................... 63

6.3 Further work .................................................. 65
  6.3.1 Network architectures ..................................... 65
  6.3.2 Training .................................................... 66
  6.3.3 Data generation ............................................ 66
  6.3.4 Stability and Behavior verification ....................... 67
  6.3.5 Implementation and test on a real system ................ 67
Chapter 1

Introduction

Model predictive control (MPC) is a control paradigm that originated in the process industry. It is notable for an inherent ability to handle both constraints and multi variable systems. The basic idea of model predictive control is simple and intuitive. Let us assume that the system that should be controlled is in state $a$, and we want it to go to state $b$. If we have a model of the system, we can then try to find a series of inputs that takes the system to $b$. The first of these inputs is then applied to the real system, we wait for one time step, observe the new state of the system and repeat the same process over again.

With all these advantages, why are MPC not used everywhere? There are two major drawbacks of MPC controllers compared to traditional proportional, integral and derivative (PID) controllers. The first is the need for a model of the system to be controlled. While having a model can be beneficial when designing a PID controller, it is often possible, and less time consuming to tune the controller to the real system. The second drawback, and the one that is relevant for this thesis, is the computational demand of MPC controllers. A PID controller only requires a few operations per control cycle and can even be implemented without any microprocessors. MPC on the other hand requires a much more demanding optimization problem to be solved each cycle.

The computational requirements are part of the explanation to why MPC originated in the process industry. When the system under control is a slow chemical process with control cycles spanning several minutes, it does not matter if it takes a few seconds to solve for the next control input. In the beginning of this century, the vast majority of applications for MPC was still in controlling large plants, mostly refineries and in the chemical industry. Since then a dramatic shift has occurred. More powerful hardware and better solvers have enabled the use of MPC in faster systems and on embedded hardware. New applications that can take advantage of MPC has started showing
According to Ferreau, Almer, Peyrl, et al. [3], this trend has been most noticeable in the automotive industry, but also in such diverse fields as aerospace, medicine, robotics, and power electronics. Here MPC’s ability to handle constraints are used to increase efficiency, reduce wear, and improve reliability.

In some of these applications sampling intervals shorter than 1 ms are required. Solving the optimization problem that quickly is difficult. With a method known as explicit MPC (eMPC) the control outputs can however be pre-computed and stored. The controller can then find the correct output in a lookup table, foregoing the requirement to solve the optimization problem online. The disadvantage with eMPC is that both the pre-computation time and the storage space scales exponentially with the number of constraints and the length of the time horizon. Many variants of sub-optimal eMPC have been suggested to solve this problem.

A similar, but not directly related, idea is to train a neural network (NN) to imitate the output of a solver. In academia this idea first showed up in the beginning of the nineties, not long after MPC itself started to gain academic traction. While this period was the start of academic study of MPC, the interest in using neural networks was lukewarm, with, at best, a few papers published per year. Interest from the industry seems to have been mostly nonexistent until very recently. Likely goaded by the success of neural networks in other fields, such as image recognition and reinforcement learning, several papers on the subject was published last year and industrial interest has started to show up both in the form of scientific reports and master’s thesis suggestions.

The field of neural network MPC (NN-MPC) is however still very new, and there are very few guidelines for many of the implementation aspects.

1.1 Problem formulation and limitations

The purpose of this thesis is to explore some of the issues that can arise when attempting to implement NN-MPC and to evaluate if some of the recent trends within other applications areas of neural networks can be applied to the case of NN-MPC.

The study focuses on methods that use imitation learning i.e. where the neural networks is trained to imitate an “expert” which output is considered the ground truth. The expert in this case is a (possibly non-linear) MPC solver.

---

1 Sometimes called “direct neural network MPC” to diverse it from methods where a neural network is used as a model for the system, and the solution to the optimization problem is then found by a traditional solver.
Methods that use reinforcement learning for similar applications might be mentioned in relation to their network architectures, but the training procedure are not brought up.

The focus of the evaluations is on network architecture, including size, activation functions, the output layer and regularization methods. Data generation and training is mentioned when necessary for the implementation. A brief evaluation of problems that could prevent NN-MPC from being implemented in real applications is also carried out, and as such problematic behavior of the controllers is discussed when encountered. The thesis discusses, but does not attempt to prove stability; not for the expert and not for the NN-MPC controllers.

The evaluations are performed through a literature study and by implementing and testing NN-MPC controllers on a simulated system. The system is a nonlinear model of an inverted pendulum, and the expert a non-linear MPC solver. The task of the solver, and thereby the NN-MPC controller is solely to stabilize the nominal system. The controllers have to deal with constraints on both the input signal and the state. There are however no evaluations of the controller’s performance under sensor noise, disturbances or similar.

1.2 Thesis outline

Chapter 2, “Background”, covers topics required to understand the rest of the thesis. Basic theory on model predictive control and neural networks, with a focus on explaining concepts that are used in this report. Chapter 3, “Related work”, handles previous work within NN-MPC. The Chapter covers general history of the subject, previously used network architectures, how training data has been generated and some problems that has been pointed out previously.

Chapter 4, “Method” describes how NN-MPC was implemented in this thesis. It also goes into some details about the inverted pendulum and some of its complications. Chapter 5, “Experiments and Results” covers various experiments carried out to evaluate different aspects of NN-MPC controllers. The chapter starts with a section which describes the setup that was used for all experiments. Each following section then starts with a motivation for why the experiment was conducted, followed by a description of the method that was used. The result of the experiment is then reported and discussed. Chapter 6, “Conclusions and Further Work” summarizes the results and conclusions. The ethics and implications of implementing NN-MPC is discussed and directions for further work within the area are suggested.
Chapter 2

Background

Some background on neural networks and model predictive control might be needed to understand the rest of this thesis. There are several thorough descriptions on both subjects, some of them cited in this chapter. This chapter therefore attempts to focus on the concepts that are important for the rest of this thesis, sometimes leaving out details and forgoing mathematical rigor.

Section 2.1 is about model predictive control (MPC). The section starts by formulating the model predictive control problem and explains how it is usually solved. Some variations of stability conditions for MPC are then handled, and the section ends with a description of explicit MPC. Section 2.3 then treats neural networks (NN), more specifically dense feed forward neural networks. It describes general properties of NN, different variations and how they are trained.

2.1 Model Predictive Control

As stated in the introduction, the principle behind model predictive control is to use a model of a dynamic system to control its real counterpart. By using the model, a series of outputs that drives the system from the current state, to a desired state can be found. In order to find this sequence of control inputs the MPC problem is formulated as an optimization problem as can be seen in
eq. (2.1) \[eq. (2.1) \]

\[
\min_u J(x, u, k) = \sum_{t=0}^{N-1} l_k(x_k, u_k) + F(x_N)
\]

s.t.
\[
\begin{align*}
x_0 &= x_{\text{init}} \\
x_{k+1} &= f(x_k, u_k) \quad k = 0, \ldots, N-1 \\
x_k &\in X_k \quad k = 1, \ldots, N-1 \\
u_k &\in U_k \quad k = 0, \ldots, N-1 \\
x_N &\in X_f
\end{align*}
\]

Here the objective is to find a series of control inputs \( u_0, u_1, \ldots, u_{N-1} \) that minimizes some cost function \( J \) over \( N \) time steps. The cost function represents a trade-off between minimizing the deviation from 0 while also avoiding excessive use of the control. \( J \) is in turn defined by other functions: the stage cost, \( l_k(x_k, u_k) \), which defines the cost at each time step, and the terminal cost \( F(x_N) \) which is added to approximate the costs that will occur beyond the time horizon. Both categories of functions should be positive definite. The optimization problem is subject to starting at the initial state, enforced by \( x_0 = x_{\text{init}} \) and have to follow the system dynamics, represented by \( x_{k+1} = f(x_k, u_k) \). The state has to stay within the state constraints, defined by \( X_k \) and the input has to remain within \( U_k \) at each step. Finally the state has to end up within a terminal set \( X_f \) at time \( t = N \). Together with the terminal cost the terminal state constraint plays an important role in ensuring stability of the controlled system, which will be discussed further later.

\[
\min_u J(x, u, k) = \sum_{t=0}^{N-1} (x_k^T Q x_k + u_k^T R u_k) + x_N^T Q_f x_N
\]

s.t.
\[
\begin{align*}
x_0 &= x_{\text{init}} \\
x_{k+1} &= Ax_k + Bu_k \quad k = 0, \ldots, N-1 \\
x_k &\in X \quad k = 1, \ldots, N-1 \\
u_k &\in U \quad k = 0, \ldots, N-1 \\
x_N &\in X_f
\end{align*}
\]

The formulation in eq. (2.1) is very general. It covers nonlinear and time varying dynamics; time varying constraints on both state and control; and the cost function is not guaranteed to be convex. Such problems are not guaranteed
to have unique solutions, and it can be very difficult to avoid local minima. To make the optimization problem easier to solve, it is often formulated as in eq. (2.2): with linear time invariant (LTI) dynamics and a quadratic cost. The cost function is then defined by three matrices: the state cost matrix $Q$, the control cost matrix $R$ and the terminal cost matrix $Q_f$. It is important that $Q$ is positive semi definite, while $R$ and $Q_f$ are required to be positive definite.

The linear dynamics is defined by the discrete system matrix $A$ and the input matrix $B$. The state constraints $\mathcal{X}$ should be a convex and closed set, the control constraints $\mathcal{U}$ should be convex and compact, and both sets should contain the origin $\mathcal{X}$. The terminal set $\mathcal{X}_f$ should of course be a subset of $\mathcal{X}$.

Ideally the time horizon $N$ would be infinite, and under some circumstances this is actually possible. For LTI systems with quadratic cost and no other constraints than the system dynamics, this problem has a closed form solution: A static feedback law called the Linear-Quadratic Regulator (LQR)\footnote{Also known as $H_2$ optimal control.}.

The LQR feedback law can be calculated as in eq. (2.3). Where $R$ is the control cost matrix, $B$ is the input matrix for the discrete system and $P$ is the cost matrix for a given state such that $x_0^T P x_0 = \sum_{t=0}^{\infty} x_t^T Q x_t + u_t^T R u_t$ where, $Q$ is the state cost matrix.

$$K = R^{-1} B^T P$$

The value of $P$ can be calculated by solving the discrete algebraic Riccati equation (DARE)\footnote{A very apt acronym for an equation that looks like a failed attempt to list the better part of the alphabet} eq. (2.4). In this equation $A$ refers to the discrete system matrix.

$$P = A^T P A - (A^T P B)(R + B^T P B)^{-1}(B^T P A) + Q$$

The $P$ matrix has an important use even for constrained controllers. An MPC controller can be made to act exactly as an LQR controller beyond the calculation horizon, by setting $Q_N = P$, assuming that there are no active constraints beyond that point. This simplifies stability analysis a great deal.

However, once constraints on the state or output signal enters the picture, this is no longer the case. One complication that deserves special mention is that the problem might no longer be feasible. If a problem is not feasible there is no control sequence that can fulfill all the constraints, and the solution is therefore not defined. There are some ways to alter a problem to allow for more feasible solutions. The most common method is to soften the constraints by severely penalizing violations instead of outright forbidding them. This
penalty is achieved by introducing slack variables to the cost function and rewriting the constraints to include the slack variables. This method works well when the real system has limits that ideally should not be reached, but that are not truly impossible.

2.1.1 Stability of MPC

There are four categories of methods for guaranteeing stability for systems under MPC control (Mayne, Rawlings, Rao, et al. [4], which also contains a much more in depth survey of stability for constrained MPC). The four categories are:

- Terminal equality constraints
- Terminal cost function
- Terminal constraint set
- Terminal constraint set and terminal cost function

The terminal equality constraint, first analyzed by Smith, Kindermans, and Le [5], is very simple. The terminal set is simply chosen as $\mathcal{X}_f = \{0\}$ (and the terminal cost to $F(x_N) \equiv 0$). If 0 can be reached within the time horizon, this method guarantees stability. That is however a big if, and a long time horizon might be necessary to make the optimization problem feasible with this constraint.

Terminal cost methods attempt to guarantee stability by setting an appropriate terminal cost. They thereby avoid the feasibility problems associated with defining a terminal set. Grüne [6] has done a survey on this type of stabilization on non-linear systems.

Terminal constraint set methods stabilize the system by defining a terminal set where another controller, that has stability guarantees, can take over from the MPC controller. These methods are similar to the first category, but can remain feasible with shorter time horizons since the terminal set is larger and can therefore be reached in fewer steps.

The final category of methods combines a terminal cost with the terminal constraint approach. Instead of letting a second controller take over in the terminal set, the terminal cost is formulated so that the system is guaranteed to be stable within the terminal set. One example of such a method is to set the terminal cost equal to that of an LQR controller, and then choosing the terminal set so that this LQR controller can be shown to stabilize the system.
there. If the system is linear this terminal set can be calculated without much problem, for nonlinear systems however, it can be complicated.

2.1.2 Solving the optimization problem

One of the major drawbacks of MPC is that the optimization problem given in eq. (2.1) has to be solved at every time step. For linear time invariant (LTI) systems the optimization problem is relatively easy to solve if the constraints are linear and the cost function is quadratic. In that case the optimization problem belongs to the class of quadratic programming (QP), for which there exists efficient solvers. If any of those criteria are not fulfilled, especially if the system is nonlinear and cannot be readily linearized, finding a solution to the optimization problem can become a lot more time consuming and less predictable. The problem might then no longer be convex, which means that there are no longer a single global minimum. There might exist several local minima, in addition to the global minimum. The global minima might also no longer be unique or in the worst case it might not exist at all. If a solution for such a problem is found, it can often not be guaranteed to be globally optimal. Which solution that is found might also change depending on how the solver is initialized.

2.2 Explicit MPC

The most prominent solution to control even faster systems, or to lower the hardware requirements, is explicit MPC. As shown by Bemporad, Morari, Dua, et al. [7] all solutions to a QP MPC problem can be exactly described by a piece-wise affine polyhedral function. That is: The feasible set can be split into regions and in each such region the optimal control output is an affine function of the state. It is therefore possible to state the control input as an explicit function of the current state, which is why these methods are referred to as explicit model predictive control. This function can then be calculated offline, possibly on more powerful hardware than what will be available to the controller. The controller then only has to do a look up to find the appropriate output, without having to solve an optimization problem. For small systems with few constraints, explicit MPC controllers can usually reach sampling times of a few milliseconds. Larger systems with more constraints and longer time horizons causes some problems for explicit MPC. The number of polyhedral regions formed can depend exponentially on both the number of
constraints and the prediction horizon \[8\], causing rapidly growing requirements on both storage space and inference time. In some scenarios explicit MPC controllers can even have a worst case evaluation time that is worse than for the corresponding implicit controller \[9\]. To avoid this growth various sub-optimal versions of explicit MPC has been developed \[8\]. Speed and storage requirements vary, but for some approaches it is possible to reach sampling times as small as tens of nanoseconds when implemented directly in hardware, as done in Bemporad, Oliveri, Poggi, et al. \[10\].

Just as their implicit counterpart, explicit MPC controllers can handle hard non-linearities, such as state and output constraints, very well. With an extension called hybrid MPC, by Bemporad and Morari \[11\], MPC controllers can also handle discrete states and systems with different, but still locally linear, behavior in different regions of the state space. The drawbacks are that the complexity of the controller increases even further, and the design process requires an intimate knowledge of the system. If the system dynamics are not well-understood (e.g. when MPC is applied to a non-linear black box model) or has smooth but not negligible non-linearities, explicit MPC will be hard to apply.

### 2.3 Neural Networks

Artificial neural networks (often only referred to as just neural networks) are a broad category of trainable computational models inspired by biological systems. Neural networks consist of “Neurons” (sometimes referred to as units), that can send information to each other through one way connections. One of the most basic types, and the type most relevant for this thesis, are dense feed forward neural networks. Such a network is divided into layers, each with a number of neurons. Feed forward refers to the fact that the neurons in a layer only have connections to neurons in the next layer. There are no connections within a layer or to previous layers, so information can only flow forward. The word “dense” just indicates that every neuron in a layer is connected to all the neurons in the next layer.

In practice a neural network is implemented through vector and matrix operations. Each layer takes a vector as input and first performs a matrix multiplication, where each element in the matrix represents the weight of a connection. A bias vector is then added and finally a so called activation function is applied to the result before sending it to the next layer. The whole operation
is described in eq. (2.5).

\[ x_{n+1} = y_n = \sigma (A_n x_n + b) \]  

(2.5)

In this equation \( A_n \) is a weight matrix, containing the trainable parameters for the layer, \( b \) is the trainable bias vector. The bias is used to adjust the activation point of \( \sigma \), the activation function. The variable \( x_n \) is the output from layer \( n \), and \( y_n \) is the output which is then fed to the next layer as \( x_{n+1} \).

There is a biological intuition for each part of this operation. Each layer can be viewed as having \( M \) neurons, where \( M \) is the number of rows in the matrix \( A_n \). The values in each row can then be interpreted as how that neuron weighs the input from each neuron in the previous layer. The input itself is represented by \( x_n \). The clever part is the activation function and the bias. Biological neurons do not just perform a weighted sum of their input, instead they only activate when that weighted sum reaches a critical point. The \( \sigma \) function represents how the neuron activates and, together with the bias, it decides when the neuron activates. The process together with the corresponding mathematical operations are illustrated in fig. 2.1.

![Figure 2.1: An illustration of the principles behind a dense feed forward neural network.](image-url)

Mathematically the activation function and the bias is what distinguishes
the operations in a layer from a simple matrix multiplication. While a matrix multiplication can represent any linear function, the operations in a single layer can approximation any continuous function on a compact subset of $\mathbb{R}^{M_{n-1}}$.

The layers of neurons that are neither the input or output layer is referred to as hidden layers. All the networks as defined here has at least one hidden layer, and more layers can easily be added to create deeper networks.

Deeper networks have been shown to have more expressive power than shallower ones with the same number of neurons. For rectifier linear unit networks (explained in section 2.3.2), it has been shown by Montufar, Pascanu, Cho, et al. [13] that the number of linear regions that such a network, with $N$ hidden layers of $M$ neurons each and an input layer with $M_{in}$ neurons, can represent, scales is bounded by

$$\left( \prod_{k=1}^{N-1} \left[ \frac{M_k}{M_{in}} \right]^{M_{in}} \right)^{M_{in}} \sum_{j=0}^{M_{in}} \binom{N}{j}$$

and therefore scales with

$$\Omega \left( \left( \frac{M}{M_{in}} \right)^{N-1} M_{in} M^{M_{in}} \right).$$

The number of represented regions is in other words polynomial in the width of each layer, but exponential in depth. Deeper architectures do, however, suffer from the vanishing gradient problem, described by Hochreiter [14], which limits the practical gain of just adding more layers. Increasing depth might also have other drawbacks. Klambauer, Unterthiner, Mayr, et al. [15] notes that for a varied set of regression tasks the most successful NN models were often relatively shallow, using 2 or 3 hidden layers, despite deeper models being tested. This lack of performance gain could indicate that there are some other drawbacks of increased depth, e.g. loss of numerical precision or an increase in local minimums in the loss function, but it might also be a result of insufficient training data to make use of the increased expressive power of the deeper models.

### 2.3.1 Training Neural Networks

Training neural networks is usually done in a supervised manner, meaning that training requires data that consist of inputs, often referred to as examples, paired with the correct output, usually called labels. Training a network is

---

3This terminology stems from classification problems, where neural networks have had the most successful applications.
essentially trying to solve an optimization problem. The weights of the connections between neurons should be adjusted so that the difference between the networks output and the corresponding label is as small as possible. The size of this difference is defined by selecting a loss function, $L$, and the output from the loss function is often referred to as “loss”. Which function that is selected depends on the purpose of the neural network and the nature of the data. A common choice for regression tasks is the mean square error (MSE), while classification networks requires loss functions that can handle probabilities.

There is a slight caveat to the goal of the training process: The network should be able to generalize, i.e. to correctly predict the output even for inputs close to but not in the training data. This requirement means that weights that give a higher loss on the training data can sometimes be preferable if those weights give a lower loss on a similar data set that the network has not been trained on. The data set is therefore split into two subsets: a training set that is part of the optimization process, and a validation set that is used to continuously test how the network generalizes.

The actual training process for a network starts with initializing the weights of the connections between the neurons, usually according to some random distribution. The following three steps are then repeated over and over until the training is either stopped manually or according to some criteria:

1. Forward propagation
2. Backward propagation
3. Weight update

The forward propagation step simply consists of feeding the network training examples and comparing the networks output with the labels. How the error, i.e. the difference between the network’s output and the labels, is measured is defined by a loss function $L$. The measurement is often referred to as the loss.

The backward propagation step then calculates the gradient of the loss as a function of the weights in each layer. This is an iterative process, starting with the output layer and working its way backward. The iterations are necessary since the loss gradient of a layer is dependent on the gradients of all the layers after it. During the backward propagation step there are two common problems that get more pronounced when the number of layers increase: vanishing and exploding gradients. The problem of vanishing gradients was analyzed by

\footnote{\textit{e.g.} the root mean square error, but there are many other ways to define this function.}
Hochreiter [14]. In this paper it was shown that under some circumstances the loss gradient will decrease exponentially for each layer, and after a couple of layers it will be so small that no meaningful learning can occur. Exploding gradients are essentially the opposite problem. The gradients instead start to increase exponentially.

In the weight update step the gradients are used to update the weights. This update can be very simple, such as in gradient decent which uses

\[ w_{\text{new}} = w_{\text{old}} - \alpha \cdot \delta \]

where \( w_{\text{new}} \) is the updated weights, \( w_{\text{old}} \) is the previous weights, \( \delta \) the gradient and \( \alpha \) is a hyper parameter known as the learning rate that decides how much the weights are updated in each step. Many other update rules exist but most that are used for training neural networks are variations of gradient decent. One such variation is gradient decent with momentum, suggested by Rumelhart, Hinton, and Williams [16]. In this algorithm a term calculated from previous gradients is added when updating the weights, giving the parameters something similar to the momentum of a physical particle. A more complicated update rule, and the one most relevant to this thesis is Adam, by Kingma and Ba [17].

The training procedure described above is not exact and since the optimization problem it is trying to solve is not convex, prone to get stuck in local minimums. To avoid getting stuck a technique known as mini batching is often used. Mini batching introduces a stochastic element to the training process by splitting the data set into several smaller batches. A forward, backward and update pass is then performed with the data in a mini batch before moving on to the next. The batch size is another hyper parameter that might have to be adjusted. If the batches are too large the training could easily get stuck. If the batches are too small the training process can become unstable. Another important drawback with small batches is that training takes more time, due to more updates having to be calculated, and that the architecture of modern GPUs is better suited for calculating large batches.

The takeaway from this section should be that training neural networks with mini batches and randomly initialized weights is a stochastic process and the results may vary even with identical hyper parameters.

\[ \text{The second reason is probably not noticeable for large neural networks with large examples, but can become a major issue for smaller networks as will be demonstrated in section 5.2.} \]
2.3.2 Activation functions

In the early days of artificial neural networks the activation function was often modeled as a step. Later on it was replaced by smoothed variation like the logistic function and the hyperbolic tangent (tanh) function, due to requirements from the training algorithm. Lately rectified linear functions (ReLU) \cite{18}, and variations there of has come to dominate the field.

\[
ReLU(x) = \begin{cases} 
  x & \text{if } x > 0 \\
  0 & \text{otherwise}
\end{cases}
\]  

(2.7)

The rectified linear function is very simple, if the input to the neuron is positive, the output is the same as the input. If the input is negative, the output is zero, as described in eq. (2.7) and illustrated in fig. 2.2.

\[
LeakyReLU(x) = \begin{cases} 
  x & \text{if } x > 0 \\
  0.01x & \text{otherwise}
\end{cases}
\]  

(2.8)

The ReLU function does however have a slight problem: If the weights for a neuron happen to be in a state so that there are essentially no input that will make the neuron activate, the error gradient will also be zero during back propagation, meaning that the neuron will remain inactive, and effectively dead. To avoid this problem the leaky ReLU function was introduced by Maas, Hannun, and Ng \cite{19}. The leaky ReLU function lets some negative output thorough, giving the neuron a chance to recover from what would otherwise be a dead state eq. (2.8).

\[
selu(x) = \lambda \begin{cases} 
  x & \text{if } x > 0 \\
  \alpha e^x - \alpha & \text{otherwise}
\end{cases}
\]  

(2.9)

Another interesting activation function is the scaled exponential linear unit (SELU)\cite{15}, which is described in eq. (2.9). Networks with SELU activation functions will normalize their weights during training, and thereby avoids most of the problems with vanishing and exploding gradients. The parameters \(\alpha\) and \(\lambda\) are calculated to give the weights a desired mean, \(\mu\) and variance, \(\nu\). The most common case is \(\mu = 0\) and \(\nu = 1\) for which the corresponding parameter values are \(\alpha \approx 1.6733\) and \(\lambda \approx 1.0507\).

Note that the SELU function requires a calculation of an exponential when the input is negative. This calculation means that the SELU function takes more time to evaluate than the ReLU function, which only requires a conditional evaluation. In a large network these calculations will have a marginal
effect compared to the matrix multiplications, but for small networks that might be implemented on embedded hardware it could become an issue. Since whether or not the calculation has to be performed is conditional based on the input value it could create some uncertainty in the inference time of the network, which is not ideal for control applications.

The last activation function described here is the hyperbolic tangent function. This function is rarely used as an activation function anymore but was used in several of the early attempts at Neural Network MPC and is still often used to limit the output from the last layer of a network.

![Activation Functions](image)

Figure 2.2: Plots of the activation curves of ReLU (top left) leaky ReLU (top right), SELU (bottom left) and the hyperbolic tangent function (bottom right).

### 2.3.3 Batch normalization and dropout

Batch normalization and dropout are two methods commonly applied to neural networks for image classification. Their prevalence in image classification makes them interesting target for investigation for applications in NN-MPC. Dropout is a regularization method first suggested by Hinton, Srivastava, Krizhevsky, *et al.* [20] and later described by Srivastava, Hinton, Krizhevsky, *et al.* [21]. The principle of dropout is simple. Nodes are turned off, or “dropped” at random during training. The idea is that a neural network trained with dropout should act more like an ensemble of several smaller networks, which
in turn should prevent overfitting and improve generalization. Dropout is only active during training. For inference all nodes are active.

Batch normalization is slightly more complex. The idea is to normalize the mean and variance of the input to each layer. During training the normalization takes place over each mini batch, and during inference, a learned mean and variance is used. There are several potential benefits: it prevents vanishing and exploding gradients, acts as a regularization method and allows for efficient training with significantly higher learning rates. The idea was presented by Ioffe and Szegedy [22] and has since then reached a widespread use.
Chapter 3

Related work

The first suggestion to replace an on-line MPC optimizer with a feed forward neural network was made by Parisini and Zoppoli [23]. The principle is simple: A dense feed forward neural network is trained on data pairs consisting of a state and an optimal control input that has been calculated by a solver.

Strictly speaking this approach is an explicit control method, just like the linear explicit controller, since the neural network defines a function from the state to the control input. However, to avoid confusion and to keep with common terminology it will be referred to as neural network MPC (NN-MPC). In [23] there is also a theoretical result that shows that feed forward neural networks can in fact be used in this way. Based on [25] it shows that a sufficiently large feed forward neural network is able to approximate a non-linear model predictive controller (NMPC) to an arbitrary degree under the assumption that the non-linear MPC policy can be defined as a function. This assumption was later proven to be true by Boom, Botto, and Hoekstra [26], under the condition that the optimization problem has a unique solution.

Perhaps the most interesting property of neural networks mentioned in [23] is their ability to, under some conditions, avoid the so-called curse of dimensionality, in this case referring to the exponential growth of complexity with the number of constraints and a longer prediction horizon. They do however note that the unpredictable nature of a trained process makes using this approach on real unstable systems difficult. The authors also present examples of simulated systems that are successfully controlled by NN-MPC. The networks used are shallow, even if the authors suggest earlier in the paper that deep networks might be used, uses tanh activation units and are trained by

\footnote{This paper does however not mark the beginning for the application of neural networks in control. An early survey of NN applications was made by Antsaklis [24] in 1990}
using gradient decent with momentum.

The main advantage of NN-MPC over explicit MPC is that the same framework can be used for nonlinear problems without modification. Neural networks ability to avoid exponential growth in storage requirements while retaining good control performance has also been noted. For example, in [27] an NN-MPC controller is shown to perform within 1.5% of the explicit controller that was used to train it, despite using less than a hundredth of the storage space. Karg and Lucia [27] also note that the time that a NN-MPC controller requires for calculating the control signal is independent of the state, which is very convenient for control applications.

3.1 Generating and selecting training data

There are many different approaches to generating training data. The simplest method, that is used by Parisini and Zoppoli [23], is to define a region in the state space where the approximating controller should be able to operate, and then uniformly sample states from that region. Data pairs are then generated by having an “expert”, in the form of a solver for optimization problems, solve the problem at each of the sampled points. However, unless the sampling region is carefully defined, optimal trajectories might start in the region but then venture outside where no training data is available.

Another approach, common in imitation learning, is to sample states from entire trajectories resulting from when the system is controlled by an expert. By doing this, the region where training data is generated should be more relevant to where the system ends up, but this approach is not without its own problems. If the approximating agent (here it would be a NN-MPC controller) generates a slightly sub-optimal trajectory the system might stray further and further into uncharted territory, resulting in a gradually larger divergence from the expert’s trajectory. To fix this issue, Ross, Gordon, and Bagnell [28] suggest that trajectories generated by the agent should be used instead, after first training the controller on a few trajectories generated by the expert. Using this method should increases the likelihood that there will be data points near trajectories that the agent ends up on. A variation of this method was used by Ericson [29]. In this variant expert trajectories are occasionally started from states on a trajectory generated by the NN.

For nonlinear optimization solvers a good initial guess can drastically improve the computation time. In [30] a radial basis function (RBF) network is used to approximate the control signal. The output from the network is not
good enough to control the system satisfactory when compared to solving the problem online. However, using the networks output to generate an initial guess reduced the online calculation time required by the solver by an order of magnitude. This approach avoids the unpredictability of using a trained controller directly; the result will always be at least as good as calculating the solution on-line. Despite decreasing the execution time, this method is still significantly slower than using a neural network directly and the time for finding a solution is still uncertain. Aside for controlling real systems this method could possibly be used for speeding up the generation of training samples by using guesses from the partially trained neural network to initialize the optimization process.

### 3.2 Stability of NN-MPC

One of the major drawbacks of NN-MPC is the lack of methods for analyzing stability properties of the resulting closed loop system, as noted by Parisini and Zoppoli [23], Bemporad, Oliveri, Poggi, et al. [10], and Karg and Lucia [27]. One way to approach the stability problem is to make sure that the expert controller stabilizes the system, even when bounded errors are introduced into the control output. If the difference in output between the expert and the NN controller can be shown to remain within these bounds the system should also be stable under control of the NN.

This approach was taken by [31]. Here the authors present a training method for neural networks that can give a probabilistic upper bound on the maximum error of the neural network compared to the optimal solutions. This bound is used in combination with a robust controller formulation that guarantees constraint satisfaction with the neural network. Interesting to note that this approach is not limited to neural networks, but to any learning system. The main drawback of this method is that the training process is prohibitively computationally expensive. In an example with only two states the bounds has to be relaxed slightly for the problem to be tractable, and even then the training required 500 hours on a quad core CPU.

Probabilistic guarantees might also not be enough for neural networks. As first shown by [32], neural networks used for image classification can be susceptible to adversarial attacks. It does not seem unlikely that this could also be the case for NN-MPC networks. On the flip side there are a growing amount of literature on how methods to counter these adversarial examples [33] in classification networks, some of which might be useful in further work on NN-MPC.
Another way to handle the stability problem is to analyze the trained NN independently of the expert. Moriyasu, Ueda, Ikeda, et al. \cite{34} show local stability around a set point by numerically calculating the Jacobian of the neural network and linearizing the system. Local stability is, at best, a first step towards verifying stability in a region, but it could be an interesting tool for finding points in the state space where more training is necessary.

A third approach to stability is to perform additional computations to ensure that the control signal fulfills some criteria. Both Chen, Saulnier, Atanasov, et al. \cite{35} and Karg and Lucia \cite{27} present very similar ideas where the output from the neural network is projected onto a control invariant set, represented by a polytope. In doing so they can ensure that the state for a LTI system remains feasible. The method is presented as a way to ensure that the state remains feasible for all further controls. However, since the set of allowed states is a polytope, and therefore bounded the method also ensures that the state is bounded which in turn implies stability in the most general sense. The system will however not necessarily be asymptotically stable. This method has two computational drawbacks. The invariant set must be defined and calculated beforehand, and the step projections step required a constrained LQ optimization problem to be solved online at each time step. The optimization problem is only in the current state and control, so it is likely to be much smaller than the corresponding MPC problem for a implicit controller, but it still requires the solver to be present.

3.3 Neural network architectures for NN-MPC

The architecture of the neural network defines the non-trainable aspects of a neural network: the number of neurons, how they are connected, which activation functions they use etc. Choosing a suitable architecture for implementing a NN-MPC controller is an important part of the design process. There are however no hard rules for how to do it. This section is a summary of the architectures that has been used in previous work, together with some theoretical results that might act as guidelines when choosing an architecture.

3.3.1 Network size and structure

There are few guidelines for how to choose an appropriate network sizes for a NN-MPC controller, something which is also true for most other application domains. For MPC controllers using linear dynamics there is however one
theoretical result that can be used. As mentioned in the background chapter, section 2.2, it is a well-known result that the solution of any model predictive control problem for a LTI system with a quadratic cost function and linear constraints, can be turned into an explicit function of the current state and that function is represented by piece-wise affine regions [7].

Both Karg and Lucia [27] and Chen, Saulnier, Atanasov, et al. [35] notes that the structure of the eMPC function overlaps with the functions that can be represented by ReLU networks. Therefore, eq. (2.6) from [13], can be used to calculate a network size that can be guaranteed to be able to represent a given eMPC controller. Also, recollect that the number of regions that a ReLU network can represent grows exponentially with the number of layers, implying that these networks could be an efficient way to represent eMPC controllers.

The use of eq. (2.6) for calculating the required size of the network comes with several limitations. The equation can be used to show that a network is large enough to exactly replicate a control policy by making sure that the lower boundary for the number of affine regions that it can represent is higher than the number of regions in the corresponding explicit MPC controller. However since the equation only gives a lower bound, applying it naively can result in larger networks than is required and, it does not account for the fact that some approximations might be allowable or even desired. Furthermore, this method is not applicable to non-linear systems or networks with other activation functions than ReLU. Finally, a network’s ability to represent a sufficient number of affine regions does not necessarily translate into the ability to reliably learn a control policy.

Chen, Saulnier, Atanasov, et al. [35] use eq. (2.6) to calculate the number of regions, and ends up with a network consisting of 2 layers with 8 neurons each for controlling a double integrator. In a second experiment a four state constrained LTI system is successfully controlled with a three layer network with 16 neurons per layer. Unlike in the first experiment this size is not based on use of the equation, since calculating the explicit MPC controller was deemed ‘‘computationally burdensome’’. No motivation for the network size used in the second experiment was provided.

The theory appears to be of limited use when deciding the network size and structure. It would be appropriate to look at how these parameters have been decided in previous work. The first paper to use dense feed forward networks for NN-MPC, by Parisini and Zoppoli [23] uses networks with a single hidden layer, but mentions that deeper networks could be an option. In their first experiment with a double integrator the layer has 40 neurons. In their second experiment, controlling a spaceship in 2D with six states and two inputs, the
layer had 120 neurons. Results were promising, but there is no mention on how these number were found.

When looking at methods used in earlier work the selection process for network size is often only mentioned briefly and only occasionally with a motivation. In [27] a full explicit controller is compared to a small feed forward neural network (6 layers deep with 6 neurons per layer) and some other approaches to lower the memory footprints of explicit controllers. The size of the network is selected by comparing the performance of several different networks with the same memory footprint. It does however not mention which sizes, except for a single layer network with 43 neurons, which performs significantly worse than the deeper network.

The test compares the average settling times of the approximations relative to the full explicit controller. The neural network performs within 1.5% of the full controller despite having a memory footprint that is more than two orders of magnitudes smaller. This result also indicates that the lower bound given by eq. (2.6) is very pessimistic for small networks (with less than two times as many neurons per layer than there are states). The full explicit MPC controller has 2317 regions, while the formula suggests that the network should be able to represent at least 57. Unfortunately, there is no comparison of evaluation time or worst-case error. This paper, by Karg and Lucia [27], does however mention one interesting quality of neural networks: Since the operations executed in a forward pass is always the same, independent of the input, the evaluation time should be almost constant.

Lucia and Karg [36] chose the network size by testing different networks with the same number of neurons. The best structures that they found were both fairly deep architectures, one with six layers of 15 nodes, and one with nine layers of ten nodes, both having the same mean square error (MSE) on the test data. A single layer network with the same number of neurons had a MSE that was three times higher, and a two layer network had about 50% higher MSE. The number of neurons is not necessarily a good way to limit the search for network size. Both the inference time and memory footprint is largely determined by the number of connections and not the number of neurons.

A peculiar structure with 2 neurons in the first layer, and 50 neurons in the following two layers were used by [31]. There is however no motivation for this structure or mention if other variations were tested.
3.3.2 Network types

There are some sources that use different neural network architectures than dense feed forward nets. The most common is probably radial basis function (RBF) networks [37]. These networks differ significantly from feed forward network in both structure and training methods. They were first used for NN-MPC by Neumerkel, Franz, Kruger, et al. [38] and have later been used by Csekő, Kvasnica, and Lantos [39] and Stogiannos, Alexandridis, and Sarimveis [30]. RBF networks are widely different from feed forward neural networks in how they are implemented and trained. Together with the relatively limited success in learning MPC policies in the last paper, these differences mean that they will not be further studied here.

Kumar, Tulysan, Gopaluni, et al. [40] uses a combination of a recurrent neural network (RNN) and a long-term short-term memory (LSTM) network to control a small LTI plant. While this plant is not that interesting for MPC control, since the system is linear and lacks constraints, the results indicate that a LSTM network alone have problems stabilizing the plant at all. The RNN performs better, but still not nearly as well as the initial controller and fails to eliminate steady state errors. The combination, both networks linked by a single layer network, performs much better and can also handle steady state errors. A likely cause for the RNN’s failure to handle the steady state error is that it lacks access to the integrated error. The LSTM network is probably able to learn this through its long-term memory mechanism, but it would likely be computationally cheaper to include this error as another state and give that information to a feed forward network or RNN.

3.3.3 Activation functions

There is some variation in which kind of activation function that have been used previously. Early implementations like in [23] tend to use hyperbolic tangent or logistic activation functions, while more recent work implementations (e.g. [27] and [34]) use variants of ReLU activation [18]. Up until very recently there were no comparisons at all of the different types of activation function for the specific task of NN-MPC, so it is likely that this trend is mostly a result of the success of ReLU activation within other areas of deep learning. Now there is however a recent comparison of networks with ReLU and tanh activation by Andersson and Näsholm [42], which concludes that the ReLU networks had better performance and required significantly less training time when imitating a constrained linear controller.

The treatment of the output layer also varies. Both some older articles
like Parisini and Zoppoli [23] and newer e.g. Ericson [29] use a hyperbolic tangent function to limit the control signal. Others, such as Hertneck, Köhler, Trimpe, et al. [31] and Karg and Lucia [27] use linear output layers, while Boom, Botto, and Hoekstra [26] use a linear output layer with a hard limit.

### 3.3.4 Batch normalization and dropout

Very few earlier studies appear to have used these methods. Only Ericson [29] appears to have used both, while Moriyasu, Ueda, Ikeda, et al. [34] only use batch normalization. Since most of the work in NN-MPC predates both methods it is not surprising that they have not been very prominent earlier but even in more recent work they are not nearly as common as in other neural network applications. Neither does there appear to be any evaluations of how these methods affect NN-MPC networks. The paper that use them does not do any comparisons.

### 3.4 Reductions in calculation time

One of the reasons to use NN-MPC is to lower the time required to calculate the time input value on a given hardware. Some of the earlier work includes reports on measured times for NN-MPC controllers which could give a clue to which speedups that may be expected. Moriyasu, Ueda, Ikeda, et al. [34] have a total calculation time of 18 ms on a Xenon processor at 2.6 GHz. Of that time only 0.022 ms was from the neural network. The remaining time was spent calculating the unscented Kalman filter used for state estimation. Lucia and Karg [36] do a comparison of the calculation time of the expert and a nine layer network with ten neurons per layer on an Intel i7 at 3.5 GHz. The NN took 0.07 ms compared to the experts 200 ms, a speedup of about $10^3$. They also implemented the same network on a Cortex M0+ microcontroller. On that chip one sample interval took 37 ms, which is still 5 times faster than the expert running on a much more powerful processor. Perhaps the most relevant comparison was done by Andersson and Näsholm [42], where a NN-MPC controller is compared to a fast implementation of an approximating online solver. The NN averaged 21.8 μs compared to the fast solver which required 560 μs for a time horizon of 10 steps. The NN-MPC controller was also judged to give better output than the solver. If the time horizon was increased to 60, the same as the expert, the calculation time rose to over 18 000 μs.
3.5 Terminology

There is no consensus on what to call Neural Network MPC. Notable variants include "Neural Regulator" [23], "Neuro optimizer" [43] and "Neuro-control" [38]. Others, such as Karg and Lucia [27] and Andersson and Näsholm [42] do not name the method at all.

Some confusion can also occur between NN-MPC as described here, and approaches that uses a neural network to model the system in a traditional MPC controller. Sometimes these methods are also called neural network MPC, (e.g. [44]) or the similar sounding DeepMPC (eg. [45]). Adding to the confusion it is not uncommon for the two approaches to be used in conjunction (e.g. [34], and [46]), and [30] even defines using a NN to approximate the controller as direct NN-MPC and methods that uses a NN to approximate the system models as indirect NN-MPC.
Chapter 4

Method

This chapter gives a general description of the methods that were used in this thesis to implement the NN-MPC controllers so that they could be evaluated. To implement NN-MPC three things have to be defined: a reference system to control, a method for generating training data and the architecture of the neural network together with its training parameters. The reference system is described in some detail since it has several pitfalls that make it complicated to control. The method for generating training data is a simple uniform sampling, with some additional steps to ensure that the sampling region is approximately equal to that where trajectories will pass through. Finally, a short description of which kind of network architectures that were studied and the training method that was used.

4.1 Reference system: Inverted pendulum

An inverted pendulum on a cart was used as a reference system. Inverted pendulums are a classic example of an unstable, non-linear system. The system, illustrated in fig. 4.1, consists of a pendulum with a mass at its end, connected to a cart through a joint. The system can be controlled via a force acting horizontally on the cart. The goal of the regulator is to move the cart to a reference point while maintaining the pendulum in an upright position. The state space formulation that was used can be found in eq. (4.1). This formulation assumes that the inertia of the rod holding the pendulum and the friction between the ground and the cart is negligible. It does however account for a friction in the
joint between the pendulum and the cart.

\[
\begin{align*}
\dot{x}_1 &= x_2 \\
\dot{x}_2 &= \frac{-mg \sin x_3 \cos x_3 + mlx_3^2 \sin x_3 + \mu \theta mx_4 \cos x_3 + u}{M + (1 - \cos^2 x_3)m} \\
\dot{x}_3 &= x_4 \\
\dot{x}_4 &= \frac{(M + m)(g \sin x_3 - \mu \theta x_4) - (lmx_3^2 \sin x_3 + u) \cos x_3}{l(M + (1 - \cos^2 x_3)m)}
\end{align*}
\]  

(4.1)

\begin{enumerate}
\item \(x_1\) is \(x\), the position of the cart
\item \(x_2\) is \(\dot{x}\), the velocity of the cart
\item \(x_3\) is \(\theta\), the angle of the pendulum
\item \(x_4\) is \(\dot{\theta}\), the angular velocity of the pendulum
\item \(l\) is the length of the pendulum
\item \(\mu_f\) is the friction coefficient for the pendulum joint.
\item \(M\) is the mass of the cart.
\item \(m\) is the mass of the pendulum.
\item \(g\) is the gravitational acceleration.
\item \(u\) is the control signal, the force applied to the cart.
\end{enumerate}

When the angle of the pendulum is close to zero, the system is approximately linear. However, at larger angles a higher control signal is required to elicit the same response in angular velocity from the pendulum. When the angle goes towards \(\pi/2\) radians the required force on the cart goes towards infinity, and the system eventually becomes uncontrollable when the pendulum is parallel to the ground. If the control signal is constrained to a limited range
the increased force required will result in an “angle of no return” where it is no longer possible to push the pendulum back towards a zero angle. If the control signal is constrained to \( \pm u_{\text{max}} \), this angle, \( \theta_{\text{max}} \) is given by eq. (4.2).

\[
\theta_{\text{max}} = \arctan \left( \frac{u_{\text{max}}}{(M + m)g} \right)
\]  

(4.2)

The calculation for \( \theta_{\text{max}} \) assumes that the angular velocity at that point is 0 rad/s. If the angular velocity is directed away from 0 the angle where recovery is impossible will be smaller, i.e. the angle of no return \( \theta_{\text{max}} \) is a function of both the angle \( \theta \) and the angular velocity \( \dot{\theta} \). This connection creates a complication when trying to control the system. An aggressive controller that does not account for the angle of no return can easily drive the system into an uncontrollable state. Another way to formulate this statement is that the amount of available control authority affects the size of the feasible set in the angular and angular velocity dimensions. This restriction means that a controller that does not account for these restrictions can fail to stabilize the system, even if it has local stability guarantees.

4.2 Generation of training data

Data sets were generated by uniformly sampling a set, referred to as the final set, in the state space. This set was determined by first selecting a target set where the resulting NN-MPC controller should perform well. The target set should include the origin\(^1\) and ideally be invariant\(^2\) under control of the expert. If it is not and there are trajectories that leave the target set, there needs to be training data along those trajectories. This is the reason for the final set. Ideally the final set would be the minimal control invariant set that contains the entire target set. To approximate this set trajectories were first generated from uniformly sampled points in the target set and the final set was then chosen as a n-dimensional parallelepiped enclosing all trajectories.

An example, generated for the inverted pendulum, of how the trajectories and the initial points of the final training set can look like can be seen in fig. 4.2 and fig. 4.3. These figures also illustrate why the data points generated by the trajectories were not used as training data: The distribution gets a strong bias

\(^1\) Depending on the solver that is used as an expert more constraints like requirements that the target set is recursively feasible might apply.

\(^2\) An invariant set \( \mathcal{X} \) is a set such that \( x_0 \in \mathcal{X} \Rightarrow x_t \in \mathcal{X} \forall t > 0 \), for a given dynamic system, i.e. if the state enters the set, it will never leave. Obvious examples: The event horizon of black holes and Hotel California
towards the origin. As can be seen in fig. 4.3, the method used here also has a disadvantage in that it is a very coarse approximation of the desired final set. As a result many of the data points might never be reached from the target set, even with a sub-optimal controller. This means that time will be wasted both on generating these points and on training the network on them. The network might also end up learning how to handle states that it will never encounter in its intended application, possibly at the cost of lowering performance in more relevant states.

When the data points had been determined they were then labeled by letting the expert calculate the optimal control output for each sampled point. The expert used was a non-linear solver with quadratic weights for the state and control cost. The terminal cost was calculated by solving the DARE, as shown in eq. (2.4) on page 6.

Trajectory based methods were not used for several reasons. Just using trajectories from the expert on the inverted pendulum results in a large concentration of examples near the origin, as can be seen in fig. 4.2 and fig. 4.3. Using agent-based trajectories to get data points causes even more problems, especially for the purpose of this thesis. While these methods are interesting and have solid reasoning behind them, they could cause trouble if applied to an unstable system. If the controller fails to stabilize the system, data points could be generated far outside the region where any control would still be
of use. Most importantly however: These approaches make the training data dependent on the network that is evaluated. When comparing different NN-architectures it would be better if all networks could be trained on the same data set, to avoid having to deal with another variable in the evaluation. Being able to generate a single data set once also saves a lot of computations.
4.3 Network architectures and studied hyper-parameters

An important part of this thesis is to compare different NN architectures. As mentioned in the related work section, dense feed forward networks have previously been successful at learning MPC, and were therefore used in favor of radial basis function networks or RNN networks. The input to the controller networks was the current state of the system and the output was the control signal. An illustration of the control loop can be found in fig. 4.4.

The largest differences between network architectures in earlier work appears to be in network size, which activation function that are used and how the output from the network is treated. The experiments were therefore focused on these areas.

4.3.1 Training method

The method for training the networks differed from the usual in one important aspect: Instead of training the networks on one data set, they were trained on two. First the networks were trained on a small data set with small mini batches, and then on a large data set with larger mini batches. A comparison and further discussion of these methods can be found in section 5.2 on page 35.
Chapter 5

Experiments and Results

Several experiments were performed to gain a better understanding of the behavior and requirements of NN-MPC. The experiments are focused on gaining a better understanding of what works and what does not, with the focus on network architectures.

The first section describes the general setup that was common for all experiments, including parameters for the dynamic model and data generation. Each following section then describes an experiment starting with a short motivations and setup followed by the results and a discussion relating to that particular experiment.

The first experiment in section 5.2 evaluates the training method described in section 4.3.1 of the previous chapter. The following sections evaluate different network architectures (e.g. network size and activation functions), both based on how well a network could replicate the expert’s output and how well it could control the nominal system compared to the expert. Finally, the qualitative behavior of some of the controllers is evaluated by studying a couple of selected trajectories.

5.1 Experimental setup

This section gives a detailed description of the general experimental setup and methodology. It also includes parameters describing where training data was generated and how the networks were trained.
5.1.1 Data generation

All data sets were generated with the method described in section 4.2. The target set was selected as

\[
\chi_{\text{init}} = \left\{ x \mid s.t. \ |p, \dot{p}, \dot{\theta}_4| < 0.2, |\theta| < 0.3 \right\}
\]

which is plotted as an orange rectangle to the left in fig. 4.2. Based on the extent of the trajectories the resulting final data set was chosen as a linear transform of the unit cuboid, defined by the matrix

\[
D = \begin{bmatrix}
2 & -2 & 0 & 0 \\
2 & 2 & 0 & 0 \\
0 & 0 & 0.35 & 0 \\
0 & 0 & 0 & 1.3
\end{bmatrix}
\]

The resulting final region is illustrated to the right in fig. 4.2 is diamond shaped in the angle and angular velocity plane, and a square in the position velocity plane.

Two categories of data sets were generated. The “constrained” set where the expert had constraints on the pendulum angle set to \( \theta \in [-0.2, 0.2] \), and one “unconstrained” set where these constraints was not present. For both data sets the same final region was used and the output signal was constrained to \( u \in [-10, 10] \).

Dynamic system

The parameters for the inverted pendulum were chosen as follows:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>( l )</td>
<td>1 m</td>
</tr>
<tr>
<td>( \mu_f )</td>
<td>0.01 N rad(^{-1}) s</td>
</tr>
<tr>
<td>( M )</td>
<td>2 kg</td>
</tr>
<tr>
<td>( m )</td>
<td>1 kg</td>
</tr>
<tr>
<td>( g )</td>
<td>9.81 m/s(^2)</td>
</tr>
</tbody>
</table>

These parameters results in a angle of no return

\[
\theta_{\text{max}} \approx 0.3275 \text{ rad} \approx 18.8^\circ
\]

\(^1\)In case of the initial point being outside of the constraints, it appears that the expert ignored that constraint until it was again inside the feasible set. This creates some uncertainty in the behavior of the expert, but a better solution, i.e. using slack variables to create better defined behavior, could not be implemented in time.
Table 5.1: Solver parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time step $\Delta T$</td>
<td>0.02 s</td>
</tr>
<tr>
<td>Time horizon $N_t$</td>
<td>160 steps</td>
</tr>
</tbody>
</table>

calculated with eq. (4.2).

The dynamics of the system was represented by discretizing the state space equation for the inverted pendulum, eq. (4.1), with the fourth order Runge-Kutta method (RK4), i.e.

\[
\begin{align*}
  k_1 &= f(x_t, u_t)\Delta t \\
  k_2 &= f(x_t + \frac{1}{2}k_1, u_t)\Delta t \\
  k_3 &= f(x_t + \frac{1}{2}k_2, u_t)\Delta t \\
  k_4 &= f(x_t + k_3, u_t)\Delta t \\
  x_{t+1} &= x_t + \frac{1}{6} (k_1 + 2k_2 + 2k_3 + k_4).
\end{align*}
\]

(5.1) \quad (5.2) \quad (5.3) \quad (5.4) \quad (5.5)

The output $u_t$ was held constant during each sample interval, i.e. zero order hold was used.

**Expert implementation and parameters**

The expert used to label the data was an inner point optimization solver implemented with Risbeck and Rawlings [48] and Andersson, Gillis, Horn, et al. [49]. The parameters for the solver are shown in table 5.1.

The state cost coefficient matrix $Q$ for the solver was set to 10 for the position and the angle, and otherwise zero. The control cost coefficient $R$ was set to 1. The terminal cost was calculated by solving the DARE, with the given cost coefficients and the system linearized around 0. The solver was initialized by using the LQR controller with the same weights.

It should be noted that it would be hard to prove that this control formulation can be guaranteed to stabilize the system due to the lack of terminal constraints. The large time horizon and LQR terminal cost do however make stability very likely. The solution to the optimization problem should ideally be unique. This is likely for this system but will not be proven.
5.1.2 Training

Unless otherwise noted in the experiment descriptions, the method described in section [4.3.1] was used to train the networks. The small data set had $3 \times 10^4$ data points and the large data set had $3 \times 10^5$ data points. The small data set used batches of $10^3$ data point each, and the large data set used batches with $10^4$. Both sets were split into a training set and an evaluation set with 70% of the data used for training, and 30% used for evaluation.

The learning rate was initially set to $10^{-2}$ and then halved every 1000 epochs. The training data was re-shuffled at the beginning of each epoch so that the batches differed between epochs. Training of a network was stopped if the loss on the validation set did not decrease in 400 epochs or a total of 10,000 epochs had passed. The final evaluations were performed with a separate data set consisting of $3 \times 10^4$ data points. The networks were implemented and trained with Pytorch 1.0 [50].

5.2 Divided training

To evaluate the mixed training method described in section [4.3.1], four groups consisting of 15 networks each were trained with different combinations of batch sizes and data sets. All networks used ReLU activation and linear outputs. The first group was trained according to the described method: first with small batches on a small data set, and then on large batches on a large data set.

The three other groups were used as controls. One control group was trained on small batches with the large data set. A second control group was used to investigate the performance of networks trained on large batches on the large data set. The third and final control group was only trained on the small data set with small batches to check that the extra training on the large data set actually improved performance. The training parameters for all groups can be found in table [5.2]. To control for the extra training session applied on the networks in the mixed group, the networks in the other two groups also went through a second round of training with the larger batch size.

All groups were trained on the constrained data set. The networks were evaluated based on their average mean square error on the test set and their training time. The networks were trained in parallel and the training time is given for the entire group.

Training was done on a PC with an six-core Intel Core i7-6850K and a Nvidia Geforce 1080Ti GPU.
Table 5.2: The training parameters for the divided training experiment.

<table>
<thead>
<tr>
<th>Group name</th>
<th>First training set (examples)</th>
<th>BS (size)</th>
<th>Second training set (examples)</th>
<th>BS (size)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mixed</td>
<td>Small $(3 \times 10^4)$</td>
<td>Small $10^7$</td>
<td>Large $(3 \times 10^5)$</td>
<td>Large $10^4$</td>
</tr>
<tr>
<td>Small BS</td>
<td>Large $(3 \times 10^5)$</td>
<td>Small $10^5$</td>
<td>Large $(3 \times 10^3)$</td>
<td>Large $10^4$</td>
</tr>
<tr>
<td>Large BS</td>
<td>Large $(3 \times 10^5)$</td>
<td>Large $10^4$</td>
<td>Large $(3 \times 10^3)$</td>
<td>Large $10^4$</td>
</tr>
<tr>
<td>Small data set</td>
<td>Small $(3 \times 10^4)$</td>
<td>Small $10^3$</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Figure 5.1: Mean square loss on the test set for controllers trained with different training methods. Small batches to the left, the mixed method in the middle, and large batches to the right. Results for the networks only trained on the small data set are not shown here since their loss was so high that the other groups would become hard to distinguish.

5.2.1 Results

As can be seen in fig. 5.1 and table 5.3 using small batches and the large data set gives the lowest loss on the test set. However, using small batches made training take more than 5 times longer than with the mixed method, as seen in table 5.4. The mixed method had slightly worse results than just using small batches, but outperformed the networks trained with large batches by a large margin, while only taking about 50% longer to train. The networks that were only trained on the small data set were far behind the other groups.

Logs from the training shows that neither the small batch group nor the large batch control groups had any significant benefit from a second round of training.

The logs from the mixed training paradigm also reveals some interesting things. The network with the best performance after both training rounds did not have the best performance after the first round. On the contrary, this net-
Table 5.3: Mean square loss on the test set for controllers trained with different training methods, sorted by the group mean loss.

<table>
<thead>
<tr>
<th></th>
<th>Small BS</th>
<th>Mixed BS</th>
<th>Large BS</th>
<th>Small data set</th>
</tr>
</thead>
<tbody>
<tr>
<td>mean</td>
<td>0.002520</td>
<td>0.002894</td>
<td>0.007307</td>
<td>0.027271</td>
</tr>
<tr>
<td>p value</td>
<td>0.029949</td>
<td>0.000006</td>
<td>0.000000</td>
<td>nan</td>
</tr>
<tr>
<td>std</td>
<td>0.000523</td>
<td>0.000359</td>
<td>0.003063</td>
<td>0.008534</td>
</tr>
<tr>
<td>min</td>
<td>0.001802</td>
<td>0.002285</td>
<td>0.004061</td>
<td>0.014502</td>
</tr>
<tr>
<td>25%</td>
<td>0.002173</td>
<td>0.002726</td>
<td>0.004852</td>
<td>0.021351</td>
</tr>
<tr>
<td>50%</td>
<td>0.002415</td>
<td>0.002936</td>
<td>0.006331</td>
<td>0.025863</td>
</tr>
<tr>
<td>75%</td>
<td>0.002803</td>
<td>0.003075</td>
<td>0.008529</td>
<td>0.031033</td>
</tr>
<tr>
<td>max</td>
<td>0.003832</td>
<td>0.003462</td>
<td>0.014009</td>
<td>0.043386</td>
</tr>
</tbody>
</table>

Table 5.4: Training time for the different network groups in the divided training experiment

<table>
<thead>
<tr>
<th>Training method</th>
<th>Training time [h:min]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Small batches</td>
<td>20:13</td>
</tr>
<tr>
<td>Mixed batches</td>
<td>3:28</td>
</tr>
<tr>
<td>Large batches</td>
<td>2:26</td>
</tr>
<tr>
<td>Small data set</td>
<td>1:09</td>
</tr>
</tbody>
</table>

work had a final evaluation score on the first round that was more than twice as high as the lowest score. Training of this network was also stopped relatively early in the first round (the best evaluation score was at round 2300).

The lowest score on the training set after the first round was also higher than the highest evaluation or test score seen after the second round for this network group.

5.2.2 Discussion

The mixed method appears to give a good balance between loss and training time and was therefore chosen to train the networks in all other experiments. Even if using small batches gave a slightly better result, the increase in training time would have made many of the experiments unpractical. It is possible that starting with small batches and then gradually increasing the batch size, as suggested by [5], could give similar results as the mixed method. This method was however not tried due to time constraints.

The extreme time increase between the large and small batches could have several explanations. Larger batches do decrease the number of update steps, but it appears unlikely that this is the only explanation. Training was done on
a GPU, and with the comparatively small networks used here, it is possible that some overhead occurred, either due to the hardware architecture or the software implementation. In other words, it might be possible to decrease the time difference, but it is unlikely that it can be eliminated or reduced to an insignificant level.

If there had been a clear correlation between the performance in the first and the second round, some networks could have been pruned before the second round. The fact that the best network after the second round was not nearly the best after the first round suggests that this is not the case, and all networks should therefore go through to the second round.

The low performance of the networks that had only been trained on the small data set suggests that adding more data can continue to improve performance even if the data set is already significantly larger than the number of parameters in the models. It also shows that large data sets might be required to give close to optimal performance for a certain network architecture, even for a relatively simple dynamic system. That the training score after the first round was noticeably higher than the evaluation score after the second round suggests that more data not only helps the networks generalize better. Having more data also makes the sampled function easier to learn.

5.3 Complexity of controllers with and without constraints

Grid search was used to get an understanding of how large a network must be in order to approximate the control policy to a reasonable degree. Networks with a depth between 1 and 7 hidden layers and a width of between 1 and 11 neurons per layer were trained. The networks used ReLU activation functions and linear output layers. One grid was trained on data from the controller without state constraints and one grid was trained on data from a controller with constraints on the angle of the pendulum. All networks in the same grid were trained on the same mini batches in the same order.

5.3.1 Result

The result for the networks replicating the experiment can be seen in fig. 5.2. Overall the bounded networks have a smaller and more consistent loss. For both the unbounded and the bounded case the root mean square (RMS) loss has a sharp drop when the number of hidden layers increase from 1 to 2. Both cases
Figure 5.2: Root mean square error for the ReLU controllers trained to imitate the controller without constraints to the left, and with constraints to the right.

Figure 5.3: Root mean square error for the controllers trained to imitate the controller without constraints to the left, and with constraints to the right.
also seem to require at least 5 neurons per layers to reliably give acceptable results. A better view of this region can be found in fig. 5.3. From this plot it is clear that the unbounded networks do not gain much from using larger networks. The constrained case does however appear to improve with both depth and the number of neurons, for the entire test range.

5.3.2 Discussion

The flatness of the loss grid for the unbounded networks indicates that the relatively small networks tested here are likely to be able to learn that control policy. The grid plot of the loss for the bounded networks is steeper, but the largest networks still reaches a loss not much higher than the best unconstrained networks. This could indicate that the largest networks in this experiment is able to learn the constrained control policy.

It is notable that the loss as a function of network size is far from monotone. A smaller network sometimes has a lower loss than a strictly larger network. The uneven performance indicates that the training is sensitive to the initial weights and which mini batches that are used.

5.4 Comparison of output clamping and activation functions

As noted in section 3.3.3 (on page 23) there is some variation in which activation functions that are used and how the output layer is treated in previous work.

Using hyperbolic tangent activation has already been compared to ReLU by Andersson and Näsholm [42], with the outcome favoring ReLU. Here the leaky ReLU function was instead compared to the slightly more recent self-normalizing exponential linear units (SELU).

Two categories of output layer were also compared, for networks with both kinds of activation function. The first type applied a trainable affine transform to the output from the last hidden layer. The output from this layer was unlimited during training but was bounded to the constraints of the system during evaluation. The intent with this arrangement was that the network should be able to learn better in regions where the output was saturated, if all output beyond the limit was not treated equally.

The second type of output layer had a hyperbolic tangent function applied after the affine transform. The output was then scaled to match the control
<table>
<thead>
<tr>
<th></th>
<th>Relu</th>
<th>Relu tanh</th>
<th>Selu tanh</th>
<th>Selu</th>
</tr>
</thead>
<tbody>
<tr>
<td>mean</td>
<td>0.002894</td>
<td>0.003451</td>
<td>0.005026</td>
<td>0.005161</td>
</tr>
<tr>
<td>p value</td>
<td>0.020698</td>
<td>0.000005</td>
<td>0.684221</td>
<td>NaN</td>
</tr>
<tr>
<td>std</td>
<td>0.000359</td>
<td>0.000803</td>
<td>0.000727</td>
<td>0.001045</td>
</tr>
<tr>
<td>min</td>
<td>0.002285</td>
<td>0.001886</td>
<td>0.003971</td>
<td>0.003566</td>
</tr>
<tr>
<td>25%</td>
<td>0.002726</td>
<td>0.003074</td>
<td>0.004481</td>
<td>0.004256</td>
</tr>
<tr>
<td>50%</td>
<td>0.002936</td>
<td>0.003391</td>
<td>0.004764</td>
<td>0.005226</td>
</tr>
<tr>
<td>75%</td>
<td>0.003075</td>
<td>0.003687</td>
<td>0.005462</td>
<td>0.006023</td>
</tr>
<tr>
<td>max</td>
<td>0.003462</td>
<td>0.004856</td>
<td>0.006422</td>
<td>0.006562</td>
</tr>
</tbody>
</table>

Table 5.5: Mean square loss for the unconstrained controllers, sorted by mean loss.

constraints of the system. The reasoning behind this approach is similar to that for the unconstrained output. Applying a hyperbolic tangent function should allow for some information to pass through even in saturated regions, since the hyperbolic tangent function is bounded and will never truly saturate. In contrast to the first approach this one will however bound the output, hopefully leaving the network with one less thing to learn.

To compare these approaches 15 networks of each combination of activation function and output layer were trained to replicate both the constrained and unconstrained case. In total 8 groups of networks were trained. All networks in this experiment had 7 hidden layers with 11 neurons each, corresponding to the largest networks in the complexity experiment in section 5.3.

The performance was evaluated by calculating the mean square loss for each network. The network types were then ranked based on the mean of that loss for the category. Student’s t-test with a significance level of 0.05 was used to test the significance of the difference between each network and the following network in the ranking. For the constrained case scatter plots showing the size of errors based on state were also created for the best performing network in each category. These plots were created in order to study if there were any correlations between the state and the size of the errors. All errors larger than 0.02 N (corresponding to 0.1% of the maximum possible error) were plotted in the angle and angular velocity plane, and the position and velocity plane.

### 5.4.1 Results

For the unconstrained case neither network category appears to have any advantage, as can be seen in fig. 5.4. This result is confirmed by the p-values in table 5.5, which indicates that there are likely no difference between the
Figure 5.4: Mean square error for the unconstrained networks to the left, and constrained networks to the right.

Table 5.6: Mean square loss for the constrained controllers, sorted by mean loss.

<table>
<thead>
<tr>
<th></th>
<th>Relu</th>
<th>Relu tanh</th>
<th>Selu tanh</th>
<th>Selu</th>
</tr>
</thead>
<tbody>
<tr>
<td>mean</td>
<td>0.002894</td>
<td>0.003451</td>
<td>0.005026</td>
<td>0.005161</td>
</tr>
<tr>
<td>p value</td>
<td>0.020698</td>
<td>0.000005</td>
<td>0.684221</td>
<td>NaN</td>
</tr>
<tr>
<td>std</td>
<td>0.000359</td>
<td>0.000803</td>
<td>0.000727</td>
<td>0.001045</td>
</tr>
<tr>
<td>min</td>
<td>0.002285</td>
<td>0.001886</td>
<td>0.003971</td>
<td>0.003566</td>
</tr>
<tr>
<td>25%</td>
<td>0.002726</td>
<td>0.003074</td>
<td>0.004481</td>
<td>0.004256</td>
</tr>
<tr>
<td>50%</td>
<td>0.002936</td>
<td>0.003391</td>
<td>0.004764</td>
<td>0.005226</td>
</tr>
<tr>
<td>75%</td>
<td>0.003075</td>
<td>0.003687</td>
<td>0.005462</td>
<td>0.006023</td>
</tr>
<tr>
<td>max</td>
<td>0.003462</td>
<td>0.004856</td>
<td>0.006422</td>
<td>0.006562</td>
</tr>
</tbody>
</table>
different groups.

The result is different for the control policy with constraints on the angle of the pendulum, visualized in fig. 5.4. Here the ReLU networks has a significant advantage over the SELU networks. On average the ReLU networks with linear output layers performed slightly better than their hyperbolic tangent counterparts. It should however be noted that the best performing network found in this experiment is a ReLU network with the hyperbolic tangent function applied to the output.

As was already suspected from the complexity experiment, it is now obvious that the unconstrained control policy was easier to learn than the constrained policy. The worst unconstrained network has a lower loss than the best constrained network, as can be seen in fig. 5.4.

There is a large spread in the loss even within the groups. Even the ReLU networks, which have the most consistent performance on the constrained data set, the worst performing ReLU network has a loss that is 50% higher than the best performing network, as can be seen in table 5.6. The scatter plots shown in fig. 5.4, shows a clear structure for both the ReLU and the SELU networks. Only the output of the networks with linear output layers are shown here since applying a hyperbolic tangent function to the output did not noticeably affect where the errors were made. In the angle plane there is a distinct outer region with no or very small errors. The SELU network appears to be more error prone in this region, with several errors over the threshold. This low error region does not correspond directly to the constraints on the angle. There are both a region within the constraints that are virtually free from errors and there is a small triangular region outside the constraints that instead appears to be the locus for most of the large errors.

For the region inside the constraints which was not free from errors, the error gradually declines towards the origin. The last tendency is more prominent for the ReLU network. In the position velocity plane does not show any sharp borders. Like in the angle plane the networks do however have a tendency to make larger mistakes further from the origin, and especially when the cart is also moving away from zero.

5.4.2 Discussion

The dominance of the ReLU networks in the constrained case could have several reasons. The hard on or off nature of the ReLU function, might be better suited for hard constraints, especially on the output signal. This explanation is backed up by the fact that the SELU networks made several small errors in
Figure 5.5: Scatter plots of all errors larger than 0.02 N for the best ReLU network (top), and the best SELU network (bottom). The left column shows the errors locations in the angle velocity plane, and the right column shows the errors location in the position velocity plane. The radius of each dot is proportional to the absolute error made at that point. Color also indicates the size of the error, with blue being the smallest and red the largest, with green and then yellow in the middle. The blue lines indicate the region where data points were generated. The orange lines in the angle plot marks the constraints on the pendulum angle.
the large saturated regions. If the SELU networks had to use a larger portion of the network to get these regions right it could explain their relatively low performance.

The fact that all groups performed about equal in the unconstrained case also supports this hypothesis, under the assumption that the unconstrained policy is less complex. The SELU networks could then afford to use a larger fraction of their neurons than their ReLU counterparts to minimize the error in the saturated regions without increasing the overall loss. If this hypothesis holds larger networks should make the SELU networks perform closer to ReLU networks in the constrained case. Another more computationally efficient solution might be to put hard limits on the output during training, which could potentially increase performance regardless of activation function, at a very small computational cost.

The structure of the angle scatter plots in fig. 5.5 shows some interesting tendencies. The outer area with very few errors presumably corresponds to where the output must be saturated to keep the pendulum within the feasible set. In this region the output would then effectively be a function of the pendulum angle and velocity, without any consideration for the position or velocity of the cart.

Even more interesting is the border region. It appears that the largest errors are made in a region just outside the constraints. This tendency could indicate that the solvers behavior in this region is ill defined. Softening the constraints might then solve this problem. The ReLU network does however not have this tendency, at least not as obvious. Which would indicate that the solvers behavior is consistent, but complex. The region corresponds to states where the pendulum is already moving towards the equilibrium. There might therefore exist outputs lower than the maximum that can take the systems to a state within the constraints within the time span of one sampling interval. The solver would then have an interval of outputs to choose from. The solver could then be consistent, but the policy might be complex or even discontinuous in this area. Both ReLU and SELU networks represent continuous functions, so a discontinuity could explain why they have such trouble in this region. Slack variables would still be a good solution, since they should make the solution continuous.

Since the ReLU Networks with linear outputs performed best on average in this experiment, they were chosen as the standard in the experiments where only one type could be tested.
5.5 Direct performance evaluation

Comparing the output of the networks to that of the expert does not necessarily say anything about how close that controller comes to following optimal trajectories. To evaluate how well each controller actually manages to control the system the networks was tasked with controlling a system from an initial point until it was settled. The cost of the resulting trajectory was then compared to that of the expert.

Each controller from the output clamping experiment was tested by initiating the system on 497 points, uniformly sampled from the target set. The trajectories were considered completed when the input $u$ was at $10^{-4}$ of the initial input. The scores, $V$, was calculated as described in eq. (5.6). The weights used by the expert controller was used to calculate a cost for each trajectory. Then the cost was divided by the cost of the expert’s trajectory. To adjust for constraint violations the sum of the violations minus the sum of violations made by the expert for that trajectory was added to the score. If a controller failed to stabilize the system from a initial point the cost for that trajectory was counted as double that of the expert. Unstable trajectories have extremely high cost and including them would make the measurement unusable. The initial points where some controllers fail are also likely to be some of the most challenging, so just excluding those costs would reward bad controllers.

The final score of a controller was calculated as the sum of the score for all trajectories.

\[
V(N) = \sum_{i=0}^{496} \min \left\{ \max \left\{ \frac{J^N(x^{(i)}_0)}{J^*(x^{(i)}_0)}, 1 \right\}, 1 + \sum_{x \in \mathcal{X}^N_i} h(x) - \sum_{x \in \mathcal{X}^*_i} h(x) \right\}
\]

(5.6)

\[
h(x = [p, \dot{p}, \theta, \dot{\theta}]) = \begin{cases} |\theta - \theta_{con}| & \text{if } |\theta| > \theta_{con} \\ 0 & \text{otherwise} \end{cases}
\]

(5.7)

$J^N(x^{(i)}_0)$ Trajectory cost when the system is controlled by the neural network $N$

$J^N(x^{(i)}_0)$ Trajectory cost when the system is controlled by the expert.

$\mathcal{X}^N_i$ The set of all states in the trajectory from initial point $i$ when the system is controlled by the neural network $N$

$\mathcal{X}^*_i$ The set of all states in the trajectory from initial point $i$ when the system is controlled by the expert

$\theta_{con}$ The constraint for the angle of the pendulum. Not present in the unconstrained case.
Table 5.7: Scores for the Unconstrained controllers

<table>
<thead>
<tr>
<th></th>
<th>Relative cost</th>
<th>Rel. cost of stable traj.</th>
<th>Stable trajectories (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Relu</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mean</td>
<td>1.012805</td>
<td>1.001443</td>
<td>98.862115</td>
</tr>
<tr>
<td>std</td>
<td>0.002634</td>
<td>0.001797</td>
<td>0.180670</td>
</tr>
<tr>
<td>min</td>
<td>1.010335</td>
<td>1.000297</td>
<td>98.393574</td>
</tr>
<tr>
<td>max</td>
<td>1.019395</td>
<td>1.007437</td>
<td>98.995984</td>
</tr>
<tr>
<td><strong>Selu</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mean</td>
<td>1.023381</td>
<td>1.008547</td>
<td>98.500669</td>
</tr>
<tr>
<td>std</td>
<td>0.012563</td>
<td>0.011660</td>
<td>0.282623</td>
</tr>
<tr>
<td>min</td>
<td>1.011682</td>
<td>1.000758</td>
<td>98.192771</td>
</tr>
<tr>
<td>max</td>
<td>1.062738</td>
<td>1.045805</td>
<td>98.995984</td>
</tr>
<tr>
<td><strong>Relu tanh</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mean</td>
<td>1.014399</td>
<td>1.001026</td>
<td>98.661312</td>
</tr>
<tr>
<td>std</td>
<td>0.002866</td>
<td>0.000516</td>
<td>0.280578</td>
</tr>
<tr>
<td>min</td>
<td>1.011090</td>
<td>1.000345</td>
<td>97.991968</td>
</tr>
<tr>
<td>max</td>
<td>1.020589</td>
<td>1.002181</td>
<td>98.995984</td>
</tr>
<tr>
<td><strong>Selu tanh</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mean</td>
<td>1.020975</td>
<td>1.003771</td>
<td>98.273092</td>
</tr>
<tr>
<td>std</td>
<td>0.003427</td>
<td>0.003085</td>
<td>0.182783</td>
</tr>
<tr>
<td>min</td>
<td>1.015997</td>
<td>1.000786</td>
<td>97.991968</td>
</tr>
<tr>
<td>max</td>
<td>1.028669</td>
<td>1.010792</td>
<td>98.594378</td>
</tr>
</tbody>
</table>

Scores from constrained and unconstrained cases are not directly comparable.

The constrained networks can generate lower scores than the expert by ignoring the constraints, whereas the unconstrained networks will always have a score that is equal to or higher than the experts. The adjusted relative cost is intended to account for this problem but there is no guarantee that the sum of the added constraint violations makes up for this advantage.

5.5.1 Results

Overall the network controllers perform well. Even on the worst measurement, the relative cost for the unconstrained controllers in table 5.7, the worst ReLU network performs within 2% of the expert.
Figure 5.6: Box plot of the performance scores of the unconstrained controllers to the left, and the constrained controllers to the right. Note that the plots in the lower row represents different measurements. The unconstrained controllers use the relative score when excluding unstable trajectories, in order to avoid the influence of the instability penalty. The constrained controllers instead use the relative cost adjusted for constraint violations.
Table 5.8: Scores for the Constrained controllers

<table>
<thead>
<tr>
<th></th>
<th>Relative cost</th>
<th>Adjusted rel. cost</th>
<th>Stable trajectories (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Relu</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mean</td>
<td>1.003252</td>
<td>1.004922</td>
<td>99.799197</td>
</tr>
<tr>
<td>std</td>
<td>0.001260</td>
<td>0.001550</td>
<td>0.075896</td>
</tr>
<tr>
<td>min</td>
<td>1.002195</td>
<td>1.003104</td>
<td>99.598394</td>
</tr>
<tr>
<td>max</td>
<td>1.006161</td>
<td>1.007905</td>
<td>100.000000</td>
</tr>
<tr>
<td><strong>Selu</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mean</td>
<td>1.017658</td>
<td>1.019671</td>
<td>99.692102</td>
</tr>
<tr>
<td>std</td>
<td>0.021177</td>
<td>0.020616</td>
<td>0.225997</td>
</tr>
<tr>
<td>min</td>
<td>1.002940</td>
<td>1.005871</td>
<td>99.196787</td>
</tr>
<tr>
<td>max</td>
<td>1.078956</td>
<td>1.079447</td>
<td>100.000000</td>
</tr>
<tr>
<td><strong>Relu tanh</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mean</td>
<td>1.003325</td>
<td>1.005062</td>
<td>99.825971</td>
</tr>
<tr>
<td>std</td>
<td>0.001977</td>
<td>0.001987</td>
<td>0.103694</td>
</tr>
<tr>
<td>min</td>
<td>1.000584</td>
<td>1.002930</td>
<td>99.598394</td>
</tr>
<tr>
<td>max</td>
<td>1.008257</td>
<td>1.009770</td>
<td>100.000000</td>
</tr>
<tr>
<td><strong>Selu tanh</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mean</td>
<td>1.013579</td>
<td>1.015115</td>
<td>99.732262</td>
</tr>
<tr>
<td>std</td>
<td>0.010892</td>
<td>0.011130</td>
<td>0.163955</td>
</tr>
<tr>
<td>min</td>
<td>1.000514</td>
<td>1.003368</td>
<td>99.397590</td>
</tr>
<tr>
<td>max</td>
<td>1.037111</td>
<td>1.039496</td>
<td>100.000000</td>
</tr>
</tbody>
</table>
As can be seen in fig. 5.6, the ReLU networks provides both better and more consistent performance compared to the SELU networks. This difference in performance is similar for both the unconstrained and constrained case. Clamping the output with a hyperbolic tangent function does not appear to affect performance noticeably for either kind of activation function.

The networks replicating the unconstrained expert have a hard time stabilizing a few points, as can be seen in table 5.7. The ReLU networks are perhaps slightly more consistent, but at least one networks from every category fails to stabilize the system in at least 5 initial states.

In the case with the constrained controller several networks manage to stabilize the system in all initial states. The scores in this case are also closer to the experts, as can be seen in section 5.5.1.

5.5.2 Discussion

The performance of the unconstrained controllers differs from what would be expected when only looking at the loss in section 5.3. In that experiment the loss did not differ significantly between any of the groups. In this direct test however the ReLU networks performed much better than either of the SELU groups. Part of the explanation for this difference could be that SELU networks might tend to produce larger steady state errors, which would be punished by this measurement method. steady state errors would however not explain why the SELU networks fails at stabilizing more trajectories than the ReLU networks. The variance in performance is also much larger for the unconstrained SELU networks than for the ReLU networks, which is also surprising when comparing to the results in section 5.3.

For the constrained networks the loss appears to be a much better indicator of performance in this experiment. When just looking at the scores it even appears that the SELU controllers performs a little bit better than would be expected from their loss. The apparently good performance might however just be an indication of that the constraint violation penalty was set too low. The fact that no SELU network managed to stabilize all trajectories supports the suspicion that these networks gained lower scores by violating constraints to a higher degree.

The fact that every group contained at least one network that failed to stabilize one or more of the initial positions was quite surprising. The constraints at \( \theta_{\text{con}} = 0.2 \) should keep the angle well away from the angle of no return at \( \theta_{\text{max}} \approx 0.32 \). This failure is worrying since it indicates that an otherwise well behaved controller can sometimes create gross constraint violations.
One interpretation of the instability problem is that the controllers are due to hesitation, which will be defined as a tendency to not apply maximum output despite this output being warranted by the expert’s control policy. Even if these errors are very small, they are biased (since the output should be saturated, mistakes can only be made in one direction) and will be integrated over the trajectory. For an inverted pendulum this effect would be even more noticeable since the control authority over the pendulum angle gets lower further from zero.

In fig. 5.5 in section 5.4 it can be seen that especially the SELU network makes a lot of small errors in the area where the output would be expected to be saturated. ReLU networks seems to be less affected by this problem. Possibly because their linear structure allows them to more easily approximate these flat regions.

The networks with hyperbolic tangent output would not have to learn how to limit themselves in the saturated regions. They would still be expected to have problems with hesitation since the output from the final layer, before the hyperbolic tangent function is applied would have to go towards infinity for the output to reach the limit. Under these considerations it would be interesting to test if the performance improves if a hard limit on the output is applied during training.

5.6 Effect of depth on performance

In section 2.3 on page 11 there is a short discussion on how depth of a network affects performance. Most regression tasks do not appear to benefit from depths beyond a few layers, but several of the NN structures in more recent work within NN-MPC has used deeper structures with up to 9 layers (i.e. [36]). From the grid experiment in section 5.3 it can be concluded that more hidden layers increases performance up until maybe 3 layers, at least for the constrained data set.

The following experiment was conducted to study if the performance continues to increase beyond those 3 layers. For each depth between 1 and 7 layers a groups of 15 networks were trained on the constrained data sets. To test even deeper network structures one group with 11 layers and one group with 16 layers were also trained. All networks had 11 neurons per hidden layer, used ReLU activation functions and linear output. To save time the 7 layer group from the clamping and activation experiment in section 5.4 were also used in this experiment.
5.6.1 Results

The results in fig. 5.7 and table 5.9 shows that there is a significant increase in performance with the addition of each layer up to and including to a depth of 7 layers. The group with 11 layers does not differ significantly from the
group with 7 layers. Training of the group with 16 layers were aborted since several of the networks failed to improve at all. This phenomenon was also observed during preliminary experiments with very deep networks. In another preliminary experiment ReLU networks with 21 hidden layers rarely managed to learn. SELU networks with a depth of 75 layers could however still be trained.

5.6.2 Discussion

In the light of previous work, the results are not surprising. They do however confirm the suspicion that NN-MPC can benefit from unusually deep structures compared to other applications of dense feed forward networks. There does however appear to be a limit to how far this increase goes. The 11 layer group if anything performs slightly worse than the 7 layer group, indicating the existence of a sweet spot. Even if there are no difference in performance between these two groups the 7 layer group would be preferable in applications due to the smaller networks lower inference time and storage requirements.

The reason for the lack of improvement between the two largest groups could have several reasons. The optimal control policy for this problem might just not be complex enough to gain anything from a network with more expressive power. It might also be the case that there is not enough data available, at least not in the right places, to learn a more complex policy. The slight decrease in performance does however indicate that there are some other phenomena to. Likely related to diminishing numerical precision.

Further experiments would be required to tell the reason for the decline in performance increase. An experiment similar to this one performed on the unconstrained data set could test if the lack of complexity is the reason. The performance would then stop increasing at a lower number of layers. The data hypothesis could be tested by once again performing the same experiment but with a smaller or larger data set. If the amount of data is the reason, the performance should stop increasing at a lower number of layers. Parts of the numerical precision hypothesis could also be tested in a similar manner, by varying the precision of the weights in the neural network. This approach would however need a lot more theoretical background.

The failure to learn phenomena indicates that there is a hard limit to the number of layers that can be used, independent on the reason for the soft limit. Due to the soft limit this hard limit is not a practical problem for this instance of NN-MPC. If further experiments would show that the soft limit can be moved beyond the hard limit for ReLU networks, SELU networks might be a viable
Table 5.10: Mean square loss for the controller in the batch normalization and dropout experiment, sorted by mean group loss.

<table>
<thead>
<tr>
<th></th>
<th>Standard</th>
<th>Batch Norm</th>
<th>Dropout</th>
</tr>
</thead>
<tbody>
<tr>
<td>mean</td>
<td>0.002894</td>
<td>0.015821</td>
<td>0.312286</td>
</tr>
<tr>
<td>p value</td>
<td>0.000007</td>
<td>0.000000</td>
<td>nan</td>
</tr>
<tr>
<td>std</td>
<td>0.000359</td>
<td>0.009077</td>
<td>0.103012</td>
</tr>
<tr>
<td>min</td>
<td>0.002285</td>
<td>0.003337</td>
<td>0.167086</td>
</tr>
<tr>
<td>25%</td>
<td>0.002726</td>
<td>0.009415</td>
<td>0.262937</td>
</tr>
<tr>
<td>50%</td>
<td>0.002936</td>
<td>0.012179</td>
<td>0.292218</td>
</tr>
<tr>
<td>75%</td>
<td>0.003075</td>
<td>0.023545</td>
<td>0.349721</td>
</tr>
<tr>
<td>max</td>
<td>0.003462</td>
<td>0.033170</td>
<td>0.591565</td>
</tr>
</tbody>
</table>

alternative.

5.7 Batch normalization and Dropout

The popularity of batch normalization and dropout regularization warrants a test of their effect on the performance of NN-MPC. Two groups of 15 networks each were trained. All networks had ReLU activation and linear output layers. The performance was then evaluated by comparing the mean square loss on the training set of both groups to that of a group using neither method. The dropout probability was set to $p = 0.05$. Student’s t-test with a significance level of 0.05 was used to test the significance of the difference between each network and the following network in the ranking.

5.7.1 Results

The results in table 5.10 clearly shows that both methods decrease performance significantly. The best network with batch normalization was in the vicinity of the same loss as the control group, but the variance was much larger. From the logs it is apparent that the BN networks stopped training a lot sooner than those in the control groups. The best performing BN networks were also those that were trained for the most epochs. For dropout the best network has a loss that is almost two orders of magnitude greater than the average for the control group.

\footnote{The ReLU group with linear output from section 5.3 was used}
5.7.2 Discussion

Neither dropout nor batch normalization improves the results compared to not using them. In the case of dropout this result is hardly surprising. In image classification where dropout is commonly used the models have a huge number, often several millions, of trainable parameters, especially compared to the number of examples in a typical training set. Over fitting is therefore a serious issue and regularization is required. In this experiment however the networks are small with about $10^3$ parameters, compared to the training set which is more than a hundred times larger. Over-fitting should therefore not be a problem and regularization would decrease performance.

The performance of the BN networks is a bit harder to explain. BN is known to have a regularizing effect which could be a partial explanation. The fact that several of the BN networks were stopped relatively early in the training explains the large spread in performance. A more generous limit for how many epochs that can pass without any improvement might then give better results. That does however raise the question why the BN networks improved so slowly.

5.8 Qualitative behavior

To give an understanding of how the NN-MPC controllers behave and to find potential problems not apparent in previous experiment, an NN-MPC controller was tasked with stabilizing the system from three initial states. The first initial state was a small step in the position to test the behavior near the equilibrium. The second initial position was a much larger step in position, to test how well the network generalizes outside parts of the state space where it has seen training data. The last initial position was a more complex position, still within the target set but close to the limit of where the system could still be controlled. The last test was carried out to test the networks behavior in complex positions and near constraints. The values for the initial states can be found in table 5.11.

Several of the NN-MPC controllers from the activation function comparison in section 5.3 were tested. The first three results are from the most successful controller in that experiment, a ReLU network with hyperbolic tangent output. Some additional trajectories by other controllers were also plotted to demonstrate problematic behavior not apparent in the first three plots. The result was compared to the trajectories generated when the system was controlled by the expert, and by an LQR controller with the same weights.
Table 5.11: Numerical values for the three initial positions in the qualitative behavior experiment. If the initial value is not mentioned for a variable, that variable was initiated at zero.

<table>
<thead>
<tr>
<th>Test</th>
<th>Initial values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Small step</td>
<td>$p = 0.2m$</td>
</tr>
<tr>
<td>Large step</td>
<td>$p = 16m$</td>
</tr>
<tr>
<td>Complex</td>
<td>$p = -0.2$, $\dot{p} = -0.2$, $\theta = -0.3$, $\dot{\theta} = -0.07$</td>
</tr>
</tbody>
</table>

### 5.8.1 Results

Overall the results are promising. The NN-MPC controller manages to stabilize the system in all three positions. It should be noted that the LQR controller is not shown in any plot. In the small step it would be identical to the expert trajectory, and in the other two cases it did not manage to stabilize the system.

The small step trajectory, shown in fig. 5.8, is almost identical to the expert’s. The large step trajectory, shown in fig. 5.9, is not as good but still close and manages to stay within the constraints. The output from the NN-MPC controller displays an oscillating behavior at some points, which is not present in the output of the expert.

For the complex initial position, the NN-MPC controller manages to match the experts trajectory very well. There is however a small constraint violation at about 1.5 s.

Two problematic behaviors were identified in several controllers, including the controller used for the previous trajectories. The first behavior is oscillations in the input when the NN-MPC controller tries to maintain the pendulum at a constraint angle. The previous example showed some of this behavior in the large step experiment fig. 5.9. Another very clear example of this phenomena can be seen in fig. 5.11.

The second behavior is a tendency for some NN-MPC controllers to give a small steady state error, as for example in fig. 5.11.

### 5.8.2 Discussion

Overall it appears as the NN controller performs well, both inside and outside the target region. Especially the performance in the large step experiment shows that the network has managed to generalize the control strategy fairly well, considering that it has only been trained on data points up to 4 meters. The fact that the LQR controller with corresponding weights fails to stabilize
Figure 5.8: State and input trajectories resulting from a NN-MPC controller stabilizes the system from the small step (initial position 1 in table 5.11). The controller used where the most successful NN from the output function comparison experiment in section 5.3.
Figure 5.9: State and input trajectories resulting from a NN-MPC controller stabilizes the system from the large step (initial position 2 in table 5.11). The controller used where the most successful NN from the output function comparison experiment in section 5.3.
Figure 5.10: State and input trajectories resulting from a NN-MPC controller stabilizes the system from the complex step (initial position 3 in table 5.11). The controller used where the most successful NN from the output function comparison experiment in section 5.3.
the system in two of the initial states also shows the advantage of a more complex control strategy.

The constraint violation is small, and in this case the constraints were set so that small violations would not endanger the stability or safety of the system. For truly hard constraints it would however be very difficult to guarantee that a NN-MPC controller do not violate them.

The case with oscillations around constraints requires further study. When controlling a real system rapidly changing control input can cause increased wear, and in some cases, e.g. electric motors, even catastrophic failures. There are several possible explanations for this behavior. One could be that the expert’s behavior close to the constraints is simply very complex, and more data points, and maybe larger networks might be able to solve or at least mitigate the problem. It is also possible that this is an artifact of the expert’s behavior outside constraints being ill defined. In that case using soft constraints in the formulation of the MPC problem could be a solution.

The steady state errors are relatively small compared to the data set (4 m), but compared to the target set at 0.2 m they could be noticeable. It is unlikely that this phenomenon is purely due to a lack of data, since the control strategy near the equilibrium point should be very simple. It could however be an indication of that the networks used here are slightly too small for the control
policy, or that the networks stopped training before the learning rate became small enough to fine tune the equilibrium point.

Some hesitation to apply maximum output, as discussed in section 5.5 can be seen in fig. 5.10. At about 1 s the expert applies maximum output, while the NN-MPC controller gives an output just below maximum. Had this behavior occurred closer to the angle of no return, the angle would likely go out of the feasible region. The output of the NN-MPC controller with linear output does not show this behavior in fig. 5.11, for that particular trajectory.
Chapter 6

Conclusions and Further Work

In this chapter the conclusions are summarized followed by a discussion on the societal impact and ethical problems that might arise when using NN-MPC. The chapter ends with some suggestions of further work that could be done in the area.

6.1 Conclusions

Small dense feed forward networks appear to be capable of imitating MPC policies to a satisfactory degree. The networks are small enough that embedded applications could be viable. Unlike other methods for explicit MPC, very large time horizons and smooth nonlinearities does not appear to be a major problem with NN-MPC. At least for regularization tasks and with convex constraints.

Using a neural network to approximate an MPC control policy is often significantly faster than finding solutions online and, just as important, the time is constant and predictable. NN-MPC therefore seems suitable for embedded applications. It could be used both as an efficient method for representing linear MPC controllers, and for controlling systems with nonlinear dynamics.

Some conclusions can be drawn about how the network architecture affects the performance of a NN-MPC controller. ReLU or leaky ReLU appears to be the preferable activation function compared to SELU and the hyperbolic tangent function. This preference is both in regard to performance and inference time. SELU units do however enable deeper architectures and could possibly perform better if the output layer was bounded during training. Limiting the output with a hyperbolic tangent function appears to give slightly worse performance compared to having a linear output layer. The effect of having a hard
 Dropout regularization and batch normalization, two methods that are common in other neural network applications, do not improve performance for this application. Batch normalization slightly lowers performance, while dropout regularization severely hampers it.

As would be expected, adding constraints to an MPC problem means that a larger network will be required to accurately imitate the resulting control policy to the same accuracy. In most cases a deeper network can imitate a policy better than a shallower one, assuming that each network has the same number of neurons per layer. The rate of improvement does however decrease with depth and there appears to be a break point where the improvements stop entirely. The reason for this break point is not completely understood, and further work would be required to determine its cause.

The results from training two networks with the same architecture on the same data set can differ significantly. The in-group variations appears to decrease when the performance of the network increases. The ratio between the best and the worst network in a group is however likely to be significant. Training several networks of the same size is therefore a way to improve performance.

More data can significantly improve performance even if the current data set contains significantly more examples than the model has trainable parameters. The improvement does not appear to be solely from better generalization, but a larger data set also appears to make the policy easier to learn.

The lack of stability guarantees, even for a nominal systems, is a problem. Some NN-MPC controllers also shows less than ideal qualitative behaviors. Most notably oscillating output near constraints, steady state errors and hesitation, as defined in section 5.5 (on page 46).

\section{Societal Impact, Ethics and Sustainability}

Embedded applications of MPC is already making things like power transmission, cars, and wind turbines safer, more reliable and more efficient \cite{3}. Using neural networks to approximate MPC policies could both be used as a new and efficient way to implement linear explicit MPC and open up new applications for embedded MPC where linear models cannot be used.

A challenging task, and likely the one that is most relevant to the parts of society which is not involved in designing the controller, is how to verify that the controller performs adequately. What would it require before we, to take
an extreme example, would let an NN-MPC controller be used as the control in a fly by wire for an unstable aircraft?

The NN-MPC controller got a few things in its favor here. Compared to an implicit MPC controller there are no complex solvers which have to be analyzed, no uncertainty in calculation time and the NN-MPC will always, barring a hardware error, come up with an output for any input. The drawback is in the “an.” Even if we assume that the control strategy that the NN is imitating is flawless (and thereby avoids opening up another, separate group of problems), there are, to my knowledge, no method to verify that the control input is close enough to the intended for all states. Even worse, there is no commonly accepted definition of what “close enough” is. Nominal stability is of course a necessary requirement, but what else would be needed before a control engineer could comfortably sign of on the statement that a NN-MPC controller will perform satisfactorily? Some sort of robustness would likely be in there, together with some sort of performance guarantees (e.g. how fast the controller will react in certain parts of the flight envelope).

Unstable aircraft is also a good example of what can go wrong when humans start interacting with controllers. Pilot induced oscillation (PIO) is a phenomenon where interaction between the controller and the pilot causes the plane to oscillate violently, sometimes with catastrophic consequences. This problem can occur even with a simple linear controller, with a nominal system. Under those conditions the problem can be analyzed, and the controller design can be altered to remedy the problem. But for a NN-MPC controller, this or similar problems could be extremely hard to analyze, if they occur, and, unless something is done to prevent it, new similar problems could appear in a different part of the flight regime.

PIO is often a result of the pilot reflexively trying to counter act the movement of a system. The time period is too short for a higher-level understanding of the controller or the system to have much of an effect. In a slower system interpretability of the controller could play a more important role. The Chernobyl disaster would make a good example of where a slow but complex, and under some circumstances unstable [51], system failed. The failure had many sources, but can partially be attributed to the complexity of the plant and its controllers causing the human operators to make erroneous decisions. This example concerns the NN-MPC controllers in two ways. First, if an NN-MPC could enable MPC to be applied to areas where it has previously only been possible to use more traditional methods due to computational or storage lim-

\[1\] Arguably the biggest one being that a nuclear reactor that was unstable under some conditions was built in the first place.
itations. However, this would come with the cost of increased complexity of the system, potentially making it less safe.

The other matter is that an NN-MPC controller is a black box. State goes in, recommended input comes out, but it is nearly impossible for a human to interpret what happens in between. In combination with the possibility of significantly incorrect behavior from the controller, the black box nature of the controller could make a very dangerous mix. Would you, for example, rather have a controller that worked in about 90% of all cases, but you knew when it did not, or a controller that worked 99% of the time, but the errors where completely unpredictable? For a safety critical system at least, I would choose the first controller, since the dangerous states could be avoided.

For applications where a failure would have less dire consequences, it, at least according to my gut feeling, gets a bit less obvious. Say that an NN-MPC controller was implemented to control an adjustable shock absorber for a car. There are safety measures in place so that the ride characteristics can never be dangerous, but a malfunction in the controller could lead to decreased performance. How would consumers react if they knew that the controller could not really be interpreted? It is not unlikely that blame for a lot of perceived inconvenient behavior would land on the controller.

Even for these applications it would be necessary to develop tools for analyzing NN-MPC controllers. Unlike in the safety critical case however, it might be ok to just be able to say that a certain problem is very unlikely to be due to the controller, or if it is the fault of the controller, give some clues as to how the problem might be solved. Without such tools I would not want to be responsible for implementing an NN-MPC controller in a commercial product

### 6.3 Further work

Here are some suggestions on further work that either would directly extend what has been presented in this thesis, or things that I see would be required to make NN-MPC controllers a realistic option in real applications.

#### 6.3.1 Network architectures

Putting a hard limit on the output during training could improve performance. The method would also be relatively easy to evaluate. It was not done here due to time constraints.

What is the reason for the break point in improvement from depth? Lack of data, lack of underlying complexity or an issue with numerical precision?
6.3.2 Training

The high variance in loss between networks of the same architecture is frustrating. Currently the best bet is to train several networks and pick the best. I currently do not have any suggestions how this problem can be solved, except tweaking hyper parameters.

Ferreau, Almér, Verschueren, et al. [52] identifies rapid prototyping as one of the most important aspects when MPC is to be used in actual applications. The NN-MPC method described here has a turnaround time of several hours even for a relatively simple problem. Are there any ways to lower it? Currently the gain from using GPUs are noticeable but could perhaps be improved. The two round training system might also allow for some improvements. The first round probably does not have to be as long as the first. There might also be possible to prune some of the networks from the second round by finding some kind of pattern in which networks from the first training round performs well in the second round.

In image classification it is possible to pre-train networks for general image recognition. These networks can then be extended and trained for more specific tasks. Would it be possible to apply similar methods to NN-MPC? When for example changing the value of a model parameter, would a network trained on data generated with the old parameter be able to learn the control policy for the new value faster or with less data?

Due to the ability of ReLU networks to represent piece-wise affine functions it would be interesting if there was a way to construct such a network to exactly match a function, instead of having to rely on a stochastic training process. If such a method were found it would eliminate the uncertainty and make ReLU network an extraordinary memory efficient way to implement linear explicit MPC controllers.

6.3.3 Data generation

I see two major areas where data generation could be improved. The first is the definition of the set where data is generated, and the other is the distribution. The current method for deciding the set is rather primitive. It requires some manual guess work and will in many cases cover large areas that are nowhere near any trajectory that can result from the target set. Having a narrower set could both lower training time and improve performance of the NN-MPC controller. The set could for example be defined by the convex hull for the trajectories.
For most experiments in this thesis it was necessary for practical reasons to use data sets that were independent of the controllers. However, the expert’s policy might vary in complexity between different regions. Using a uniform distribution of data points would then not be ideal. It would be interesting to test methods that would generate more training data in regions with many or large errors.

6.3.4 Stability and Behavior verification

There are currently no proof of stability for NN-MPC controllers, even for nominal systems. For regulation of systems that are or could potentially become unstable this would be quite useful. It might be possible to use the fact that a NN-MPC is an explicit controller, and study it directly with Lyapunov methods.

Stability is not the only aspect of NN-MPC controller that require analysis methods. Constraint satisfaction and performance also needs to be studied. Currently there are as many ways to evaluate performance of a NN-MPC controller as there are papers on the subject. Suggesting some basic evaluation strategies would be a good first step in this direction.

6.3.5 Implementation and test on a real system

Implementing an NN-MPC controller for a physical system would be a good way to identify more practical problems.
Bibliography


