Correct-by-Design Synthesis of Neural Network Controllers

W. van der Velden

Master of Science Thesis

Delft University of Technology

Delft Center for Systems and Control

Correct-by-Design Synthesis of Neural Network Controllers

Master of Science Thesis

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

W. van der Velden

July 7, 2020

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

Abstract

The use of artificial neural networks is becoming ever more ubiquitous as the computational power available to use grows. The widespread implementation of neural networks as controllers in the field of systems and control is however being hindered by the lack of verifiability of these controllers. One type of controller that does not lack verifiability is the correct-by-design controller. The main drawback of correct-by-design controllers is that they inherently produce large data structures in order to store their control rules. In this work, a novel methodology to synthesize correct-by-design neural network controllers is presented in order to alleviate these issues. This methodology combines reinforcement learning techniques and abstraction based correct-by-design control verification techniques in order to synthesize neural network controllers that are correct-by-design. The procedure does so by alternating between a controller training routine and a controller verification routine in a system abstraction framework. This framework ensures that numerical training and verification results in a controller with formal guarantees applicable to real systems. Using the proposed methodology, neural network controllers are synthesized and verified in order to prove that the methodology works. In addition to this, the resulting correct-by-design neural network controllers are compared to conventional correct-by-design controllers in order to judge their performance and data requirements. This comparison also includes alternative structures used to store these different controllers. Based on this comparison, a conclusion is drawn regarding when to use which type of controller. The proposed methodology is implemented into a correct-by-design neural network synthesis framework called COSYNNC. This framework is intended as a basis for further research into correct-by-design neural network control and is publicly available.

Table of Contents

List of Figures

List of Tables

Preface and Acknowledgements

First and foremost, I would like to thank my supervisor dr. ir. Manuel Mazo Espinosa for his excellent guidance and for allowing me to pursue this topic. I also wish to thank all of the members of the research group who were present during the weekly group meetings and provided valuable feedback and suggestions. I would also like to thank all of the members of the examination committee for offering their time. Finally, I wish to thank my family, friends and girlfriend for their support during my entire studies.

Delft, University of Technology
July 7, 2020

W. van der Velden

Chapter 1

Introduction

The use of artificial neural networks is becoming ever more ubiquitous as the computational power available to use grows[6]. They are currently being utilized in a host of different data analytic tasks such as nature language translation[13], speech recognition[7] and image classification[17]. The acceleration of both interest and the use of neural networks can largely be accredited to the increase in the computation power available to us and the universally desirable properties that neural networks inherently possess. Amongst these properties are robustness with respect to the plant due to the learned nature and the cheap on-line evaluation of the network itself.

The widespread implementation of neural networks as controllers in the field of systems and control is however being hindered by the lack of verifiability of these controllers. The verifiability of neural network controllers is proving to be a major bottleneck as guarantees on the closed loop controlled system's behaviour are one of the primary requirements posed on controlled systems before they can be utilized in a practical capacity. It would therefore be desirable to develop a neural network training procedure that allows for the verifiability of neural network controllers. Doing so would drastically increase the deployability of neural network controllers in contexts where the use of neural networks makes sense from a control perspective.

One type of controller that does not suffer the problem of verifiability is the correct-by-design controller. There has already been quite some research into the correct-by-design synthesis of controllers and several tools such as PESSOA[23], CoSyMa[26] and SCOTS[31] have been developed for this purpose. The main drawback of correct-by-design controllers is that they inherently produce large look-up tables that prescribe the appropriate inputs for every possible state of the system given that the control specification is attainable from those states. This intuitively results in large data structures that represent these controllers. This limits their deployability in environments with limited amounts of memory. Chief amongst these environments are embedded systems which are often a point of interest for practical controller deployment. For this reason, alternative correct-by-design control structures are desired that require less data.

If the notion of the traditional correct-by-design control synthesis could be combined with neural networks, the robustness and cheap on-line computation of the neural networks could be combined with the verifiability of correct-by-design control. This would potentially allow for a significant reduction in the amount of data required to store such a correct-by-design controller and speed up the evaluation of said controller.

In this thesis, a novel method of combining correct-by-design controller synthesis techniques and neural networks will be presented. Through the use of both neural network synthesis techniques and numerical controller verification methods, a method to synthesis correct-by-design neural network controllers will be presented. In addition to this, traditional correct-by-design controllers and the neural network controllers will be compared in terms of their performance and their data requirements.

1-1 Notation

In this work, vectors will be denoted by $\vec{v}$ and matrices by $\boldsymbol{M}$. The l_2 norm of a vector will be denoted by $\|\vec{v}\|$. The cardinality of a set will be denoted by $|Z|$ where Z is a set. For scalars, the absolute value will be denoted by $|a|$ where a is a scalar. The notation $x \xrightarrow{u} x'$ denotes the evolution from a state x to x' given a function $f(x, u)$ under the input u. The behavioral inclusion of system S_1 by system S_2 is denoted by $S_1 \preceq_{\mathcal{B}} S_2$. The identity map on a set Z is denoted by $1_Z : Z \to Z$. The set $\mathbb{R}_+$ denotes all positive real numbers $\mathbb{R}_+ = \{r \in \mathbb{R} \mid r \geq 0\}$ including 0. The set $\mathbb{N}_0$ is the set of all natural numbers $\mathbb{N}_0 = \{z \in \mathbb{Z}, z \geq 0\}$ including 0. The notation $[\alpha, \beta]^n$ will denote the set $[\alpha, \beta] \times ... \times [\alpha, \beta]$ where $[\alpha, \beta]$ defines the set of all real numbers $[\alpha, \beta] = \{x \in \mathbb{R} \mid \alpha \leq x \leq \beta\}$. The set of all subsets of Z, also known as the power set of Z is denoted by 2^Z.

Chapter 2

Preliminaries and Problem Statement

This chapter will review the preliminary theory that is employed in this master thesis. First, the fundamental theory regarding neural networks will be reviewed. Then, the preliminary theory with respect to correct-by-design control will be reviewed. Finally, the problem will be stated.

2-1 Neural networks

This section will review the fundamental theory regarding artificial neural networks and several training methods that are available to them.

2-1-1 Fundamentals

The most fundamental artificial neural network, which shall from now on be referred to as simply a neural network, is the feedforward neural network. These neural networks were first envisioned by McCulloch and Pitts[24] and then formed into a proper mathematical model by Rosenblatt[29]. The model that is most often used to illustrate the idea of these feedforward neural networks is that of the multilayer perceptron neural network. This topology is just one of many neural network topologies that are available.

Definition 2-1-1 (Multilayer perceptron[6]). *The multilayer perceptron feedforward neural network is defined as a mapping function:*

$$\vec{y} = f(\vec{x}, \boldsymbol{\theta}) \tag{2-1}$$

where the input $\vec{x} \in \mathbb{R}^n$ is mapped to the output $\vec{y} \in \mathbb{R}^m$ by the neural network parameters $\boldsymbol{\theta}$. The function f represents the neural network and maps:

$$f \, : \, \mathbb{R}^n \to \mathbb{R}^m \tag{2-2}$$

based on these parameters.

This mapping can be seen as a network as it consists of fundamental layer functions that are interconnected into a series of the form: $\vec{y} = f(\vec{x}) = f_N \circ ... \circ f_2 \circ f_1(\vec{x})$. The reason these types of systems are called neural networks is that they are loosely based on the biological structure and functioning of the brain and the neurons within them. In such a network, individual neurons are connected to other neurons in order to manipulate the information as it flows through them.

Definition 2-1-2 (Neural network layer[6])**.** *The manipulation of the information that flows through a neural network for elementary neurons in a single layer is:*

$$\vec{x}_i = g(\boldsymbol{W}_i^T \vec{x}_{i-1} + \vec{b}_i) \tag{2-3}$$

where $\vec{x}_i$ is a vector representing the output of layer i, $\boldsymbol{W}_i$ is the weight matrix that represents the weighted edges between the neurons in layer i and the neurons in layer $i-1$, $\vec{b}_i$ is a vector that represents the individual biases added to neurons and $g(\vec{x})$ is a nonlinear activation function that manipulates the values of the neurons in a piecewise fashion.

More elaboration on these nonlinear activation functions and there purpose is provided in Section 2-1-2. Throughout this work $\boldsymbol{\theta}$ represents the neural network parameters that yield the neural network behaviour. The parameter $\boldsymbol{\theta}$ thus consist of all the weights and biases for every layer of the neural network.

The mapping that a neural network provides is determined by the weights and biases of that network. By appropriately picking these weights and biases a neural network can be used to approximate any function. The act of acquiring the weights and biases that yield the desired behaviour is called neural network training or neural network learning. There are a variety of training algorithms and methods, the most fundamental of which, are discussed in Section 2-1-3.

It should be noted that there is a wide host of different neural network topologies available. Arguably the most elementary of which is the multilayer perceptron. Amongst other often employed neural network topologies are convolutional neural networks[19, 18, 17], recurrent neural networks[6, 7, 13, 25] and long short-term memory neural networks[8, 35]. Each of these topologies have their own advantages and disadvantages with respect to different applications. Throughout this thesis the main focus will be on multilayer perceptron neural networks although it should be noted the presented methodology is robust with respect to different topologies.

2-1-2 Activation functions

As already mentioned in Section 2-1-1, the information that flows through a neural network is manipulated in a piecewise fashion by a nonlinear activation function $g(\vec{x})$. There is a wide variety of different activation functions which all serve different purposes. The fundamental purpose that all of these different activation functions share is in preventing the values in the neural network from 'blowing up' to $\pm\infty$. Without the use of these nonlinear activation functions the neural network could quickly become unstable and cause the output of the network to diverge away from a stable point. There are a number of different activation functions that are often utilized in the field of artificial neural networks, as summarized by Karlik en Olgac [14]. A few of these activation functions shall now be discussed.

Two often used activation functions are the unipolar sigmoid function and the hyperbolic tangent function. The unipolar sigmoid function is mathematically described as:

$$g(x) = \frac{1}{1 + e^{-x}} \tag{2-4}$$

The unipolar sigmoid function squeezes any value to a value between $[0, 1]$. In a control context, it is often used in neural networks controlling gating functions and has its use in a variety of neural network topologies. Another often used activation function is the hyperbolic tangent function defined as:

$$g(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{2-5}$$

This function outputs values between $[-1, 1]$. Both of these functions are able to serve as activation functions and exhibit advantageous mathematical properties that can be exploited for verification purposes as suggested by Ivanov et al. [10]. Their use in training is however somewhat degraded by the fact that evaluation of these functions and most notable the exponential is a computationally expensive operation.

Due to this, another, often used, activation function that has gained a lot of traction recently is the so-called rectified linear unit, abbreviated to ReLU, as described by Nair and Hinton[27]. It is defined as:

$$g(x) = \max(0, x) \tag{2-6}$$

The ReLU function does not suffer the drawback of being computationally expensive that the sigmoid and hyperbolic tangent function suffer, as it only required a comparison operation to compute. The ReLU function has also proven to be a viable substitute for the more expensive unipolar sigmoid function in a variety of examples such as in restricted Boltzmann machines[32, 6].

2-1-3 Training

As stated in Section 2-1-1, in order to get a neural network to exhibit the desired behaviour, training is required. With training, the notion of tweaking the neural network parameters $\boldsymbol{\theta}$ such that the neural network approximates a desired function $f(\vec{x}, \boldsymbol{\theta}) \approx f^*(\vec{x})$ is implied. In order to tweak these parameters, a common approach is to utilize an optimization routine. In this section, the most fundamental algorithm for finding these weights will be reviewed. Furthermore, two fundamental strategies for using this algorithm will be discussed.

Backpropagation algorithm

The most fundamental algorithm for finding the weights of a neural networks is the so-called backpropagation algorithm. This algorithm was popularized by Rumelhart[30] and still serves, although in an altered form, as the primary training algorithm in use today. The algorithm works by defining a loss function as a function of the neural network outputs $\mathcal{L}(\vec{y})$. This loss function represents how the neural network is performing on a set of training data with respect to some performance criteria. The weights are then tweaked as to minimize the loss function. Due to the large dimensionality of the space that each configuration of the neural network parameters $\boldsymbol{\theta}$ spans, it is generally not possible to find these weights directly. This has led to the utilization of an algorithm that is often employed in classic optimization: the gradient descent algorithm. In the so-called backpropagation algorithm[30], each individual weight in the neural network is tweaked using the gradient descent rule as to achieve $f(\vec{x}, \boldsymbol{\theta}) \approx f^*(\vec{x})$. The gradient descent rule for each weight and bias in the neural network is:

$$\theta_{k+1} = \theta_k - \lambda \nabla_\theta \mathcal{L}(f(\vec{x}, \boldsymbol{\theta})) \tag{2-7}$$

where λ is the gradient descent step, $\mathcal{L}(f(\vec{x}, \boldsymbol{\theta}))$ is the loss function as a function of the neural network's output, θ_{k+1} denotes the individual weight at the next time step and θ_k is the current weight. The gradient $\nabla_\theta \mathcal{L}(f(\vec{x}, \boldsymbol{\theta}))$ is found by utilizing the chain rule and propagating the partial derivate backwards through the network, from the output to the input, for each individual weight.

It is theoretically required to calculate the gradient with respect to all of the data points in order to perform a single gradient descent step. However, in practice this would be prohibitively expensive to compute. Therefore a variant of the gradient descent method is used that is called stochastic gradient descent. In stochastic gradient descent the loss is defined with respect to how the network performs on a small 'minibatch' of data points in the entire data set. The gradient is then adapted with respect to the loss function over a smaller batch of data points. This drastically reduces the required training time. This method is stochastic as it calculates an estimate of the gradient based on a sampled subset of data points from the entire data set.

Another important parameter in the backpropagation algorithm is the learning rate λ. This parameter dictates the step size that is taken with respect to the gradient in every gradient descent step. In practice this learning rate is not chosen as a fixed though small number but instead it is shrunk over time as $\lambda(t)$. By decreasing the learning rate over time, this prevents the algorithm from 'overshooting' a local minimum and enables the algorithm to end up inside such a local minimum. A number of these adaptive learning rate algorithms are available such as AdaGrad[5] and Adam[16].

Applying the backpropagation algorithm to a recurrent neural network results in the backpropagation through time algorithm. Conceptually the algorithm is the same but applied to the unfolded sequential neural network. In addition to this, care must be taken to respect the fact that the weights are being shared at multiple points in the network. This can be resolved, as suggested by Goodfellow[6], by introducing dummy variables $\boldsymbol{W}^{(t)}$ denoting the weight at every time step. These variables denote the contribution of the weights at time step t to the gradient and allows the network to respect the constraints on the weights.

Based on the definition of the loss function, two distinct types of learning can be defined: supervised learning and reinforcement learning. Both of these types of learning methods will now be reviewed.

Supervised learning

In supervised learning, a neural network is trained to approximate a desired function using a labelled set of data. The most obvious example where supervised learning is used, is in training neural network classifiers[20, 17, 4]. In a supervised learning setting, a loss function is defined based on the neural network output $\vec{y} = f(\vec{x}, \boldsymbol{\theta})$ and the desired output $\vec{d} = f^*(\vec{x})$.

The loss function originally used by Rumelhart[30] was the scaled sum of the square of the errors:

$$\mathcal{L} = \frac{1}{2} \sum_{c=1} \sum_{j=1}^{m} (y_{j,c} - d_{j,c})^2 \tag{2-8}$$

where c is an index over all the input-output pairs. The gradient descent minimization of this loss function causes the neural network to approximate the desired output for a given input.

Janocha and Czarnecki summarize an array of often used supervised learning loss functions in a classification setting[11]. These include, among others, the quadratic loss as used by Rumelhart $\|\vec{y} - \vec{d}\|_2^2$, the proportional loss $\|\vec{y} - \vec{d}\|_1$ and the often used log cross-entropy loss:

$$\mathcal{L} = -\sum_{j} d_j \log y_j \tag{2-9}$$

The log cross-entropy loss is often used for classification as it conditions the output to values between $[0, 1]$ such that the sum of the outputs is 1. It can hence be interpreted as returning a probability or certainty of a output or classification as generated by the network.

Supervised learning has the advantage of quickly being able to converge to a minimum in which the neural network approximates the desired output for the data which was used during training. This drastically decreases the required training time and the computational expenses the neural network requires to learn the function. It is also extremely straight forward to implement whenever a labelled data set is available, as it only requires the designer to specify the neural network dimensions, the loss function and the input and output data.

The disadvantages of supervised learning is that it requires a labelled set of data, preferable a large one, which is often either not available or difficult to acquire. It is also prone to mistakes, as any errors in the labelled set will also be learned by the network. The network will therefor never outperform the labelling process. Another disadvantage is that is does not necessarily learns to recognize patterns or other otherwise telling features that characterizes the input. In general, what it learns instead, is a mapping from inputs to outputs without any integrate knowledge of the input. This often causes supervised learned networks to perform badly in transfer conditions, where it is tasked with performing on inputs which were not inside the dataset. This often causes erratic behaviour, which is generally undesirable.

Reinforcement learning

In reinforcement learning, the desired output is not required in order to train the network. Instead, the network is trained by having it interact with an environment. Reinforcement learning is often applied in situations where a neural network is given control over some Markov decision process. The designer of the training procedure is required to specify a reward function $\mathcal{R}$ that will indicate whether or not the neural network is behaving well. The neural network is then trained with the goal of maximizing the reward function[12].

In reinforcement learning, a neural network is randomly initialized. The neural network will then generate outputs based on the inputs it receives from the environment. These inputs usually provide information about the environment through sensors or other means. It is therefore the neural networks representation of the state of the environment. The neural network then makes a decision on which action to take based on the state information. It does so by outputting an action probabilistically. Based on this probability, the action will be implemented. This stochastic element is what allows for the reinforcement learning approach to be utilized.

If the reward is immediately accredited, it allows the reinforcement learning algorithm to reinforce the action if the reward is positive and deter the action if the reward is negative. This reinforcing and deterring is done by computing the so-called policy gradients, which adapts the weights of the network with the goal of maximizing the expected reward. This type of immediate reinforcement learning for neural networks was first pioneered by Williams[36]. By implementing the input based on the certainty, there is also a chance that the input to the environment will be different than the one outputted by the network. This allows for exploration of the input space. If this input is successful it will be reinforced by the algorithm, hence improving the network performance. If it is not successful that input will have a decreased chance of being implemented in the future. This implementation balances the trade-off between exploitation of learned connections and exploration of the inputs in the input space.

Sometimes the reward is only rewarded after a certain sequence of inputs and corresponding environment state transitions. This case is called the delayed reward case, an example of which is Karpathy's Pong from Pixels[15]. In the example a neural network is taught to play the classic Atari game of Pong through the use of reinforcement learning techniques. A positive reward is awarded when the network scores a point against the opposing player, a negative reward if the opposing player scores a point against them. In this case, a list of states and corresponding neural network outputs need to be made such that, at the event that a reward is granted, the neural network can reinforce or deter its control policy. The 'playing' time between two consecutive rewards is called an episode. At the end of each episode, the algorithm reinforces all the state-input pairs of that episode if they led to a positive reward and vice versa.

There are a number of advantages to using reinforcement learning algorithm with respect to supervised learning. First of all, there is no need for a labelled dataset to serve as training data, the creation of which could be difficult, impractical or impossible. Secondly, the performance of the neural network is not limited to the quality of such a dataset. These aspects make it so that this particular technique is must better suited for teaching the neural network an internal model instead of simply a mapping. This results in a more properly behaved neural network in transfer conditions where the neural network is tasked with making decisions for situations it was not trained on.

There are also a few disadvantages associated with the reinforcement learning method. First of all, the designer needs to define a reward function such that the resulting network fits a specification. This is often not trivial and oversight on the part of the designer may lead to undesirable behaviour that does actually maximize the reward function. Another disadvantage with respect to supervised learning is that the training times are most often significantly longer as the training is not directed. It could take a long time for the neural network to discover inputs that result in a reward, as it simply has no notion of the appropriate behaviour at initialization. Finally, the reinforcement learning approach suffers the credit assignment problem where, in the case of delayed reward, it is unclear which inputs lead to failure or success. Hence, the algorithm may end up penalizing actions that would ordinarily have given positive results and reinforcing actions that do not benefit the neural network's performance.

2-2 Correct-by-design control

In the context of this thesis, the notion of correct-by-design control is one of the main focus points. In this section, the major concepts and relevant literature regarding correct-by-design control shall be reviewed.

As the name suggest, correct-by-design control regards the topic of creating controllers, which are by definition correct, due to the way in which they were synthesized. In order to synthesize such controllers, a framework is required which allows for the synthesis and verification of these controllers. A synthesis and verification framework, robust with respect to different classes of systems, calls for numerical methods. In order to use numerical methods to conclusively verify the correctness of system behaviour, finite abstractions have to be constructed instead of infinite systems which are in general difficult to analyse in a generic way.

2-2-1 Formal system definitions

In order to analyze systems Tabuada[33] introduces the concept of symbolic abstractions based on simulation relations. These concepts require a formal definition of systems.

Definition 2-2-1 (System[33]). *Given a system S, it is defined as a sextuple:*

$$S = (X, X_0, U, \rightarrow, Y, H) \tag{2-10}$$

where X is the set of states, $X_0 \subseteq X$ is the set of initial state, U is the set of inputs, $\rightarrow \subseteq X \times U \times X$ is the transition relation between states and inputs to new states, Y is the set of outputs and $H : X \rightarrow Y$ the output map.

In order to simplify the notation, it is useful to introduce a few operators and definitions. The post operator is defined such that:

$$\mathrm{Post}_u(x) = \{x' \in X \mid (x, u, x') \in \rightarrow\} \tag{2-11}$$

A system is called simple if the output set is the state set $Y = X$, the output map is identify 1_X and all states are admissible initial states $X_0 = X$. In that case, the system can be written as the triple $S = (X, U, \rightarrow)$.

In order to discuss the relations required to create finite cardinality symbolic abstraction, Tabuada also introduces the concept of system behaviour.

Definition 2-2-2 (Finite internal behaviour[33]). *The finite internal behaviour $\mathcal{B}_x(S)$ is defined as a finite sequence:*

$$\mathcal{B}_x(S) = \{x_0 \xrightarrow{u_0} x_1 \xrightarrow{u_1} ... \xrightarrow{u_{n-1}} x_n\} \tag{2-12}$$

of states x_i given inputs u_i for system S.

In case the sequence is infinite it is called the infinite internal behaviour $\mathcal{B}^\omega_x(S)$. These behaviours can be grouped into a set to form so called external behaviours.

Definition 2-2-3 (Finite external behaviour[33]). *The finite external behaviour of a system is defined by union the finite internal behaviour:*

$$\mathcal{B}(S) = \bigcup_{x \in X_0} \mathcal{B}_x(S) \tag{2-13}$$

The notion of finite external behaviour can be similarly extended to infinite external behaviour if the sequences of states is infinite for a given system S. The infinite external behaviour is defined as $\mathcal{B}^\omega(S) = \bigcup_{x \in X_0} \mathcal{B}^\omega_x(S)$. The idea is to create a finite symbolic abstraction of the infinite system S_a such that the behaviour of the infinite system S is included in the behaviour of the finite system $S \preceq_{\mathcal{B}} S_a$. This notion is called behavioral inclusion.

2-2-2 Simulation relations

In order to create such a finite symbolic abstraction, Tabuada introduces a multitude of relations between systems. If an abstraction can be constructed such that it adheres to a simulation relation between the infinite system and the abstraction, it implies the behavioral inclusion of that infinite system. This allows for the use of numerical methods on the finite abstraction such that any results are also valid for the infinite system. Hence this allows for numerical controller synthesis. The most fundamental of these relations is the simulation relation.

Definition 2-2-4 (Simulation relation[33]). *Consider two systems S_a and S_b with $Y_a = Y_b$. A relation $R \subseteq X_a \times X_b$ is a simulation relation from S_a to S_b if:*

- $\forall x_{a0} \in X_{a0}$, $\exists x_{b0} \in X_{b0}$ *such that* $(x_{a0}, x_{b0}) \in R$
- $\forall (x_a, x_b) \in R$, $H_a(x_a) = H_b(x_b)$
- $\forall (x_a, x_b) \in R$, $x_a \xrightarrow{u_a} x'_a$ *implies the existence of* $x_b \xrightarrow{u_b} x'_b$ *satisfying* $(x'_a, x'_b) \in R$.

In that case S_b is said to simulate S_a, mathematically denoted by $S_a \preceq_{\mathcal{S}} S_b$.

If $S_a \preceq_{\mathcal{S}} S_b$ and $S_b \preceq_{\mathcal{S}} S_a$ then the relation R is a bisimulation relation of the two systems. In the context of control, one is not only interested in the existence of an input such that the states are in a simulation or bisimulation relation but also in the actual input itself. This requirement prompts the introduction of the notion of the alternating simulation relation.

Definition 2-2-5 (Alternating simulation relation[33]). *Consider two systems S_a and S_b with $Y_a = Y_b$. A relation $R \subseteq X_a \times X_b$ is an alternating simulation relation from S_a to S_b if:*

- $\forall x_{a0} \in X_{a0}$, $\exists x_{b0} \in X_{b0}$ *such that* $(x_{a0}, x_{b0}) \in R$
- $\forall (x_a, x_b) \in R$, $H_a(x_a) = H_b(x_b)$
- $\forall (x_a, x_b) \in R$ *and* $\forall u_a \in U_a(x_a)$, $\exists u_b \in U_b(x_b)$ *such that* $\forall x'_b \in Post_{u_b}(x_b)$, $\exists x'_a \in Post_{u_a}(x_a)$ *satisfying* $(x'_a, x'_b) \in R$, *where* $U(x) = \{u \in U \mid ((x, u, x') \cap \rightarrow) \neq \emptyset\}$

Similarly to the simulation relation, if this relation is valid in both directions it is called an alternating bisimulation relation. Furthermore, Tabuada introduces the notion of an ε-approximate simulation relation which can be applied to all previous simulation relations. In an ε-approximate simulation relation the systems are required to be metric systems, meaning there is a metric $d : Y \times Y \to \mathbb{R}^+_0$. In that case, a relation is a ε-approximate relation if $\forall (x_a, x_b) \in R$, $d(H_a(x_a), H_b(x_b)) \leq \varepsilon$.

2-2-3 System abstractions

As eluded to before, in order to synthesize a correct-by-design controller a finite system needs to be created that simulates the actual system. Equipped with the knowledge of alternating simulation relations Tabuada, shows how such a symbolic abstraction can be created using Lyapunov stability notions. The idea is to use two discretization steps in the form of time sampling the evolution of the system and quantizing the state space.

Definition 2-2-6 (Control system[33]). *The control system Σ is defined as $\Sigma = (\mathbb{R}^n, \mathcal{C} \times \mathcal{D}, f)$ where $\mathbb{R}^n$ is the state space, $\mathcal{C}$ the set of all the inputs, $\mathcal{D}$ the set of all the disturbances and f the function that represents the plant.*

Definition 2-2-7 (Time sampled system[33]). *The time sampled simple system S_τ based on sampling time τ associated with control system Σ is defined as $S_\tau = (X_\tau, U_\tau, \underset{\tau}{\rightarrow})$ where:*

- *$X_\tau = \mathbb{R}^n$, $U_\tau = \{\chi \in \mathcal{C} \mid dom\chi = [0, \tau]\}$*
- *$x \xrightarrow[\tau]{\chi} x'$ if there exists $\chi \in U_\tau$, $\delta \in \mathcal{D}$ and the trajectory $\xi_{x\chi\delta} : [0, \tau] \rightarrow \mathbb{R}^n$ of Σ satisfying $\xi_{x\chi\delta}(\tau) = x'$*
- *$Y_\tau = \mathbb{R}^n$ and $H_\tau = \mathcal{I} : X_\tau \rightarrow \mathbb{R}^n$*

This time sampled system is related to the quantized abstraction $S_{\tau\eta}$ which is equivalent to the time sampled system S_τ expect for the quantized state space $X_{\tau\eta} = [\mathbb{R}^n]_\eta$ and the transition function satisfying $\|\xi_x(\tau) - x'\| \leq \eta$. Given that the control system Σ is input-to-state stable (ISS), this allows for an input-to-state stable Lyapunov function of the form $V(x) = \sqrt{x^T P x}$ where P is a symmetric positive definite matrix. This Lyapunov function entirely defines the ε-approximate bisimulation relation between $S_\tau(\Sigma)$ and $S_{\tau\eta}(\Sigma)$ through:

$$R_\varepsilon = \{(x_\tau, x_{\tau\eta}) \in X_\tau \times X_{\tau\eta} \mid V(x_\tau - x_{\tau\eta}) \leq \underline{\alpha}\varepsilon\} \tag{2-14}$$

if η satisfies:

$$\eta \leq \min\{\gamma^{-1}\underline{\alpha}\varepsilon(1 - \exp^{-\lambda\tau}), \overline{\alpha}^{-1}\overline{\alpha}\varepsilon\} \tag{2-15}$$

where λ and γ are parameters that bound the state trajectory via Equation 11.3 in the book of Tabuada and $\underline{\alpha}$ and $\overline{\alpha}$ are the outer eigenvalues. The proof of this relation is somewhat elaborate and provided by Tabuada, page 176. Although this approach does provide a ε-approximate bisimulation relation between the infinite system S_τ and the quantized finite system $S_{\tau\eta}$ it does require an ISS-Lyapunov function, which requires certain assumptions on the stability of the system. If the system is not stable, care must be taken to first stabilize it in order to utilize these techniques to construct an abstraction. This is a limiting factor of Tabuada's approach to symbolic abstractions.

2-2-4 Feedback refinement relation

One of the problems with the abstraction method as presented by Tabuada is that it requires the system to be input-to-state stable and the existence of an input-to-state stable Lyapunov function. In practice, the pursuit of these properties often entails the design and incorporation of a low-level controller which can be very challenging. This problem is alleviated by the feedback refinement relation.

Definition 2-2-8 (Feedback refinement relation[28])**.** *Given two simple systems S_a and S_b then $R \subseteq X_a \times X_b$ is a feedback refinement relation from S_a to S_b if for all $(x_a, x_b) \in R$ it holds that:*

- $U_b(x_b) \subseteq U_a(x_a)$
- $(x_a, x_b) \in R$ *given* $u \in U_b(x_b) \implies x'_a \in Post_u(x_a),\ x'_b \in Post_u(x_b),\ (x'_a, x'_b) \in R$

The difference, with respect to alternative simulation relations, is that for an input u_a there must exist an input $u_b = u_a$ such that the new states of both systems using that input are in the relation. For the feedback refinement relation, the same input, must result in new states that are in the relation. For the purpose of control synthesis, it is still vital that the infinite system is abstracted to a finite system. In order to achieve this using the feedback refinement relation, similarly to the ε-approximate bisimulation abstraction technique, the time sampled, state and input space quantizer system is considered. The sampling time is given by τ and the quantizer of the state space is given by η, the quantizer of the input space is given by η_u. The computation of abstractions that satisfy the feedback refinement relation now reduces to the over-approximation of attainable sets of the plant. The abstraction S_a of the infinite system S is characterized by a set of states consisting of hyper-intervals, which can be thought of as cells that cover the state space of the infinite system. The abstraction S_a will now be mathematically defined.

Definition 2-2-9 (Feedback refinement relation abstraction[28])**.** *Given two simple systems S and S_a with sampling time τ, a subset $\bar{X}_a \subseteq X_a$ of compact cells and an over-approximation function $\beta : \mathbb{R}^n_+ \times U_a \to \mathbb{R}^n_+$ then S_a is an abstraction of S if:*

- $\bar{X}_a$ *is a cover of X by non-empty, closed hyper-intervals and every element $x_a \in \bar{X}_a$ is compact*
- $U_a \subseteq U$
- $(x_a, u, x'_a) \in \rightarrow \ |\ x_a \in \bar{X}_a,\ x'_a \in X_a,\ u \in U_a$ *and* $(\phi(\tau, \vec{c}, u) + [\![-r', r']\!]) \cap x'_a \neq \emptyset$
- $Post_u(x_a) = \emptyset$ *whenever* $x_a \in X_a \setminus \bar{X}_a,\ u \in U_a$

where ϕ denotes the general solution of the unperturbed system, $\vec{c}$ is the center of the cell, r is the distance from the center to each edge of the cell and $[\![-r', r']\!]$ is the hyper-interval defined by the growth bound where $r' = \beta(r, u)$.

A graphical depiction of the state transition function based on the growth bound is provided by Figure 2-1.

Reissig et al. also provide a method to calculate the growth bound based on bounding the Jacobian of the function describing the system f. This calculation requires the function f to be continuously differentiable for every u. If that is the case Reissig et al. provide the growth bound using the Jacobian as:

$$L_{i,j}(u) \geq \begin{cases} D_j f_i(x,u) & \text{if } i = j \\ |D_j f_i(x,u)| & \text{otherwise} \end{cases} \tag{2-16}$$

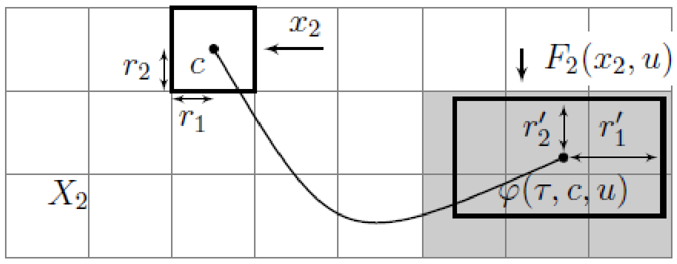

Figure 2-1: A figure showing how the growth bound is used to find the set of attainable states from the current state for a given input. The resulting set of cells that form potential transitions from the current state is colored in gray. This figure was taken from the paper by Reissig et al. [28].

with the growth bound:

$$\beta(r, u) = e^{L(u)\tau} r + \int_0^\tau e^{L(u)s} w \, \mathrm{d}s \tag{2-17}$$

where w is a disturbance term. It should be noted that there are also ways of finding a growth bound for non-continuously differentiable functions f. Using the growth bound it is now possible to construct abstractions of systems without any stability assumptions. For this particular relation it holds that $S \preceq_{\in} S_a$ or $S \preceq_R S_a$ with the exception of a state quantization function.

2-2-5 Fixed-point algorithms

The use of these relations and abstractions becomes quite apparent once it is applied to the context of control. Given that a system can be simulated using a finite system, allows for the use of numerical methods on the finite system in order to find appropriate control laws. Tabuada elaborates on how to synthesize a controller based on these relations, subject to either of two specifications: invariance and reachability. These two specifications can be further expanded using fixed-point algorithms to find more complex linear temporal logic specifications.

In the case of a controller tasked with determining inputs that will keep the state of a system in a safe set, Tabuada introduces the notion of invariance. Invariance is concerned with finding the set W for which the system will remain inside the set $W \subseteq Z$.

Definition 2-2-10 (Invariance[33, 31])**.** *For invariance, a controller is required such that:*

$$\forall t \geq t_0, \; x(t) \in Z$$

where $Z \subseteq X_a$ is the safe set. A controller that adheres to this specification can be found by iterating the operator $F_W : 2^{X_a} \to 2^{X_a}$ defined as:

$$F_W(W) = \{x_a \in W \mid x_a \in Z \, \cap \, \exists u_a \in U_a(x_a), \, \emptyset \neq Post_{u_a}(x_a) \subseteq W\} \tag{2-18}$$

Using this operator, starting with $W = Z$, results in convergence to a set $W \subseteq Z$ such that there exist inputs at all states in W that cause the system to remain in Z. This type of algorithm is called a fixed-point algorithm. From the resulting set, a controller S_c, can be synthesized that satisfies the invariance specification.

Tabuada also covers the notion of reachability. Reachability is concerned with finding the set of states for which a controller is able to control the system into a certain predefined set Z.

Definition 2-2-11 (Reachability[33, 31])**.** *For reachability, a controller is required such that:*

$$\exists T,\ x(T) \in Z$$

where $Z \subseteq X_a$ is the to be reached set. A controller that adheres to this specification can be found by iterating the operator $G_W\ :\ 2^{X_a} \to 2^{X_a}$ defined as:

$$G_W(W) = \{x_a \in X_a \mid x_a \in Z\ \cup\ \exists u_a \in U_a(x_a),\ \emptyset \neq Post_{u_a}(x_a) \subseteq W\} \tag{2-19}$$

Similarly to the invariance case, using this operator. starting with $Z = W$. will result in convergence to the set of all states $W \supseteq Z$ from which the set Z can be reached. From this set, a controller can be synthesized that satisfies the reachability specification.

By combining these specifications, the reach and stay specification can also be defined.

Definition 2-2-12 (Reach and stay[22])**.** *For reach and stay, a controller is required such that:*

$$\exists T,\ \forall t \geq 0,\ x(T+t) \in Z$$

where $Z \subseteq X_a$ is the reach and stay set. A controller that adheres to this specification can be found by first iterating the invariance operator $F_W : 2^{X_a} \to 2^{X_a}$ defined as:

$$F_W(W) = \{x_a \in W \mid x_a \in Z\ \cap\ \exists u_a \in U_a(x_a),\ \emptyset \neq Post_{u_a}(x_a) \subseteq W\} \tag{2-20}$$

followed by the reachability operator $G_W\ :\ 2^{X_a} \to 2^{X_a}$ defined as:

$$G_W(W) = \{x_a \in X_a \mid x_a \in Z\ \cup\ \exists u_a \in U_a(x_a),\ \emptyset \neq Post_{u_a}(x_a) \subseteq W\} \tag{2-21}$$

Using these operators, starting with $W = Z$ for the invariance operator F_W, followed by the reachability operator G_W with $Z = W$, results in the set W for which the system is able to reach and stay inside the set Z.

2-2-6 Tools

The systematic approach to the synthesis of symbolic abstractions and control as presented by Tabuada and Reissig is well suited to automation. A number of tools that utilize these concepts are currently available. Two such tools, that are closely related to the work presented above, will now be reviewed.

One such correct-by-design synthesis tool is PESSOA, as presented by Mazo, Davitian and Tabuada[23] and based on the work of Tabuada. PESSOA is capable of constructing finite abstractions of linear control systems and to synthesize controllers for the specifications laid down by Tabuada. This does however require some stability assumptions on the part of the plant before these synthesize techniques can be utilized.

Rungger and Zamani present another control synthesis tool, based on the feedback refinement relation presented by Reissig et al., called SCOTS[31]. SCOTS allows for the construction of symbolic abstractions of systems given that the designer provides the plant dynamics ϕ, a growth bound on those dynamics β, a bound on the disturbance w and the quantization parameters τ, η and η_u. The growth bound can be found for continuously differential plant dynamics as reviewed in Section 2-2-4. SCOTS allows for control synthesis based on invariance, reachability or even more complex linear temporal logic specifications. These controllers are then synthesized similarly to PESSOA, using the reviewed fixed-point algorithms. Both PESSOA and SCOTS allow for representing the resulting controller in the form of a reduced order binary decision diagram. The main difference between SCOTS and PESSOA is that SCOTS makes use of the feedback refinement relation, as reviewed in Section 2-2-4, in order to create the finite abstraction.

Due to the way in which fixed-point algorithm based correct-by-design control synthesize fundamentally works, the resulting controller is by definition a look-up table. This look-up table prescribes the appropriate inputs for all the states from which the specification can be adhered such that the control system adheres to the predefined specification. Since the amount of states in the abstraction grows rapidly as the quantization becomes finer, these look-up tables can quickly grow to become very large. Because of this, both PESSOA and SCOTS attempts to compress the data requirements of the resulting controller by encoding them as binary decision diagrams (as will be reviewed in Section 2-3). By describing these look-up tables as reduced order binary decision diagrams, significant memory savings and data compression can be achieved. It should however be noted that even these reduced order binary decision diagram controllers can still be quite large in terms of memory requirements and requires some overhead software to manipulate them. These factors form a limitation with respect to implementation of correct-by-design controllers on embedded hardware.

2-3 Binary decision diagrams

As already touched upon in Section 2-2-6, one of the major limitations of fixed-point based correct-by-design control synthesis methods is the resulting control structure. By the very nature of the synthesis method, the resulting controller is a (often large) look-up table. In order to compress the data, numerous tools have resorted to using binary decision diagrams to store these look-up tables. How binary decision diagrams work and how they are being used in the context of control will be discussed in this section.

Binary decision diagrams (BDD) were first introduced by Lee [21] and further studied by Akers [1] and Bryant [3] as a novel way of defining, analyzing, testing and implementing large binary functions.

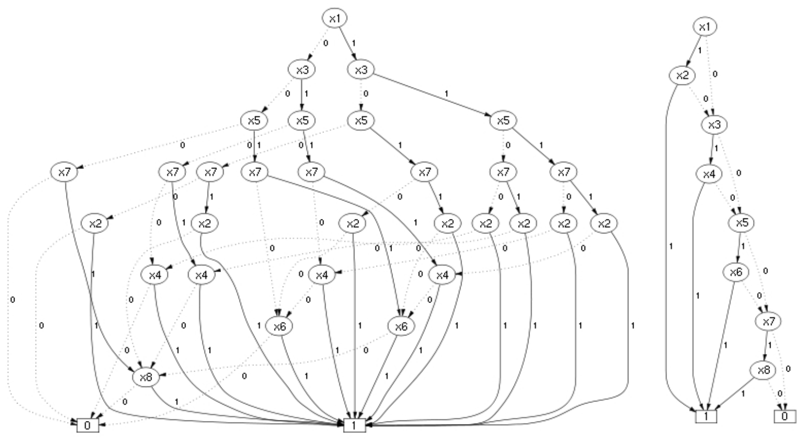

Figure 2-2: An example of a binary decision diagram and its reduced order form. The original binary decision diagram is depicted on the left. The reduced order form on the right. The figure is taken from Beyer[2].

Definition 2-3-1 (Binary decision diagram[1, 3, 21])**.** *A binary decision diagram is a rooted, directed and acyclic graph consisting of decision nodes and two terminal nodes. The graph encodes a binary function using these decision and terminal nodes. The terminal nodes label the outcome of this binary function. The binary function is of the form:*

$$f \ : \ \mathbb{B}^n \rightarrow \mathbb{B} \tag{2-22}$$

where $\mathbb{B} = \{0, 1\}$ *denotes a boolean variable.*

Binary decision diagrams are a way of encoding complex binary functions into graphs, such that the graphs can be used to evaluate binary functions. The main idea with using a graph to store a binary function, is that analysis of the graph can yield simplifications to the graph whilst still encoding the same function. In addition to this, the order of the variables in the graph can also be changed. The order of the binary variables in the graph has a large impact of the final size of the graph. Because of this, certain heuristics are often employed to find an order that minimizes the graph. By iterating these processes, the graph can be reduced in order to reduce the amount of memory required to store the binary function. Such a reduced binary decision diagram is called a reduced order binary decision diagram (ROBDD). An example of such a binary decision diagram and its reduced order form is depicted in Figure 2-2.

As hinted at in Section 2-2-6, these binary decision diagrams can be used to store correct-by-design controllers, such as the onces computed by PESSOA [23] and SCOTS [31]. This is achieved by formatting the controller in such a manner that is can be treated as a binary function. This is possible due to the fact that the abstractions, used to simulate the original to be controlled system, are of finite cardinality. Hence the state and input space of the system consist of n-dimensional hyper-rectangular cells that cover a portion of the actual spaces. These cells lay on a grid such that each and every cell has a unique n-dimensional integer coordinate in its corresponding space. It should now be noted that the correct-by-design controllers are look-up tables that map discrete state cells in the state space to discrete input cells in the input space. Since these cells have an n-dimensional coordinate expressed in integer numbers it is possible to store the binary representation of these coordinates in a binary decision diagram. The binary decision diagram of a correct-by-design controller is therefore a graph that represents a binary function that stores this mapping. This binary function is of the form:

$$\mathcal{C} \, : \, \mathbb{B}^i \times \mathbb{B}^j \rightarrow \mathbb{B} \tag{2-23}$$

where $\mathbb{B}^i$ are binary variables that represent the state space, $\mathbb{B}^j$ are binary variables that represent the input space. The function itself is a combination of binary functions that represent encoded state-input pairs denoted as $\mathcal{SI}$. Evaluating the binary function $\mathcal{C}$ with input binary variables denoting a valid state-input pair will evaluate to a binary 1 otherwise the graph will evaluate to a binary 0. Using this mechanism, the controller can be encoded in a binary decision diagram. These binary state-input pairs $\mathcal{SI}$ are encoded by taking the integer coordinates these states have in their respective space and encoding them in binary. For the state space of dimension n this implies encoding $\mathbb{N}_0^n \rightarrow \mathbb{B}^i$ where i is the number of binary variables required to encode the state space coordinates. For the input space of dimension m this implies encoding $\mathbb{N}_0^m \rightarrow \mathbb{B}^j$ where j is the number of binary variables required to encode the input space coordinates. The controller as a binary function $\mathcal{C}$ is therefore a combination of these binary state-input pairs $\mathcal{SI}_i$ of the form:

$$\mathcal{C} = \mathcal{SI}_1 \vee \mathcal{SI}_2 \vee ... \vee \mathcal{SI}_k \tag{2-24}$$

The controller can then be read out by providing it with a binary representation of the current state in which the system is and evaluating the binary function. This will result in the set of inputs that the controller prescribes.

An example of such a binary controller and state-input pair shall now be given in order to further illustrate the data structure. This particular way of encoding is the one as employed by SCOTS, but it should be noted that there are multiple ways to achieve this casting.

Example 2-3-1. *Consider a demonstrative system with a two-dimensional state space and a one dimensional input space. Now consider the domain of interest of the state space to lie between* $x_0 \in [0,1]$ *and* $x_1 \in [0,1]$ *and the quantization parameter to be* $\eta = 0.25$ *with the center of the resulting cells on the bounds. This results in both dimensions of the state space being subdivided into* 5 *parts and the total state space consisting of* 25 *cells. The point* $x = (0,0)$ *is located in the first cell with integer coordinate* $(0,0)$ *and the point* $x = (1,1)$ *is located in the final cell with integer coordinate* $(4,4)$*. Consider the system to have switched linear dynamics with two modes and the switching of these modes the control input* $u \in \{0,1\}$*. Since both dimensions in the state space are subdivided into* 5 *parts* 3 *binary state variables are required per dimension. Therefore,* 6 *binary state variables are required to encode a state cell's coordinate in the state space. For the input space only* 1 *binary variable is required since there are only two modes.*

Consider now that, based on the dynamics of an arbitrary plant, the fixed-point based synthesis algorithm returns that in state cell $x_0 \in [0.625, 0.875]$, $x_1 \in [0.875, 1.125]$ *the input to the system should be* $u = 1$*. This encodes to state cell with coordinate* $(3,4)$ *mapping to* (1)*. In binary this corresponds to* $b(3) = \{1,1,0\}$, $b(4) = \{0,0,1\}$ *to* $b(1) = \{1\}$*. Where* b *is a binary big-endian encoding function* $b \, : \, \mathbb{N}_0 \rightarrow \mathbb{B}^i$*. In order to read out what input should be given, the binary variables that represent the state coordinate should therefore correspond to the binary representation resulting in* 1*. Hence the binary state-input pair that corresponds* *to this information would become $\mathcal{SI} = \mathbb{B}_{s,0} \vee \mathbb{B}_{s,1} \vee \neg\mathbb{B}_{s,2} \vee \neg\mathbb{B}_{s,3} \vee \neg\mathbb{B}_{s,4} \vee \mathbb{B}_{s,5} \vee \mathbb{B}_{i,0}$ where $\mathbb{B}_s$ are binary variables encoding the states and $\mathbb{B}_i$ are binary variables encoding the inputs.*

This process is repeated for all the state-input pairs that result from the correct-by-design synthesis procedure, resulting in a binary function representation of the controller. This binary function can then be cast into a binary decision diagram, which can be further reduced to find the more data efficient reduced order binary decision diagram.

This particular method of casting correct-by-design controllers to binary decision diagrams is also robust with respect to nondeterminism. Meaning that multiple different inputs can be mapped to the same state cell coordinate in order to represent the nondeterminism.

As already hinted at in Section 2-2-6, the advantage of using reduced order binary decision diagrams over ordinary look-up tables is that they can significantly reduce the amount of data required to store the controller. Binary decision diagrams obtain a large part of their compressive capability from removing redundancy within the binary representation of the controller. The disadvantage to using binary decision diagrams as controllers however is that these controllers would ideally be used on embedded hardware where the amount of memory and computing power is limited. Since binary decision diagrams cannot be intuitively readout, the use of binary decision diagrams still requires additional overhead that further taxes the amount of memory and computational power on these systems.

2-4 Problem statement

In this section, the problems that can be identified based on the reviewed preliminaries are stated. Furthermore, an outline of the intended methodology is given.

As discussed in Section 2-2, correct-by-design control provides us with a generic technique to synthesize controllers which are correct-by-design. It is also robust with respect to different system classes such as linear, nonlinear and hybrid systems. Perhaps one of the most promising aspect of the correct-by-design control synthesis method is however the formal guarantees that are put on the performance of the controller once it has been synthesized. This particular feature warrants the use of correct-by-design control in a wide spectrum of applications.

However, as stated in Section 2-3, the widespread implementation of such controllers is currently being hindered by a number of problems. Most notably the large amount of data that is required to store either the raw look-up table of the controller or a binary decision diagram compressed version thereof. Even if the compressed binary decision diagrams are used, the overhead that is required to read them out and use them is still quite substantial. This leads to the following problem:

Problem 2-4-1. *Design a correct-by-design control structure that requires less data than look-up table stored correct-by-design controllers and binary decision diagram stored correct-by-design controllers.*

With regards to implementation on the memory and computational limited environment of embedded hardware, neural networks have a distinct advantage. As discussed in Section 2-1, neural networks are cheap to evaluate on-line once they are trained, as it is only a matter of performing linear mathematical operations with the exception of a nonlinear activation function. In addition to this, neural networks have already proven themselves to be capable of a wide variety of tasks including control related tasks. Encoding a correct-by-design controller as neural network could therefore be a promising lead with respect to deployment onto embedded hardware.

One of the limitations is that neural networks are traditionally difficult to verify and sporadic behaviour in vital control applications is unacceptable. It is therefore necessary to employ correct-by-design control methods in order to synthesize these neural network controllers in a verifiable way. This leads to the following problem:

Problem 2-4-2. *Design a neural network synthesis procedure that allows for the training and verification of a neural network in the role of a controller.*

An approach that one might utilize is to use an existing deterministic correct-by-design controller and try to embed the mapping that it exhibits into a neural network. This would result in a correct-by-design neural network controller if the neural network outputs the appropriate input for every state for which the controller is defined. The approach has been tried in literature [9, 34] and is indeed valid. In practice, it does however result in data structures that are of a similar size in terms of memory to binary decision diagram stored controllers.

One might argue that the reason why these neural network controllers are of equivalent size to binary decision diagram stored controllers is that the neural network is only exposed to the predefined mapping of these controllers. Since it is only exposed to the mapping it is never able to learn the intrinsic behaviour, that the plant exhibits, that actually implies that state-input mapping. By letting the neural network act as an actor within the to be controlled environment and by appropriately rewarding and deterring its behaviour as it interacts with the environment, one could teach a neural network to act as a controller. This particular neural network training method is called reinforcement learning and is discussed in Section 2-1-3.

Using this reinforcement learning technique does still result in an unverified neural network controller. Through the use of partial abstractions and fixed-point algorithms as discussed in Section 2-2, it is however possible to verify the behaviour of this controller. Furthermore, this verification can be used to further emphasize certain regions in the state space on which the neural network needs to focus during training. Through the combination of these techniques and the novel interaction between them it would therefore be possible to synthesize correct-by-design neural network controllers.

The goal of this thesis work is to explore the possibilities with regards to the synthesis of correct-by-design neural network controllers. Hence the following problem is to be solved:

Problem 2-4-3. *Given a predefined control specification, synthesize a correct-by-design neural network controller such that the resulting feedback controlled system adheres to the specification with formal guarantees.*

In order to solve Problem 2-4-3, first Problem 2-4-2 will be tackled. The neural network synthesis procedure as required by Problem 2-4-2 is explained in Chapter 3. The controllers that result from using this procedure are compared to the traditional correct-by-design controllers in order to see whether or not they are able to solve Problem 2-4-1.

Chapter 3

Methodology

This chapter will describe the methodology that is used to synthesize correct-by-design neural network controllers. The methodology works to synthesize correct-by-design neural network controllers for a variety of different classes of systems. The primary control specifications that will be discussed in this work are a subset of linear temporal logic specifications. In its current implementation the methodology supports invariance specifications, reachability specifications and reach and stay specifications. The methodology can however be extended to more general linear temporal logic specifications.

In order to describe the methodology, the primary features of the methodology will be described in a piecewise manner. First the general neural network controller topology will be explained. After that the actual synthesize procedure will be described. The synthesize procedure consists of three routines, the system abstraction framework routine, the synthesize routine and the verification routine. A figure that shows how each of these routines interact with each other is depicted in Figure 3-1. Each of these routines will be discussed in their own section. The interaction between these different routines will also be described.

The entire procedure as explained in this chapter is available as a correct-by-design neural network controller synthesize framework that has been coined COSYNNC[1]. The general use and layout of this framework will also be described.

3-1 Neural network controllers

In order to discuss the methodology that is employed to train a correct-by-design neural network controller, it is paramount to define such a controller. In this thesis work, the term neural network controller will be used to denote a neural network that is used in a feedback configuration in order to control a plant. The feedback configuration is depicted in Figure 3-4. In this section, potential neural network topologies that can be used as neural network

[1]The framework is available at: https://github.com/WardvanderVelden/COSYNNC

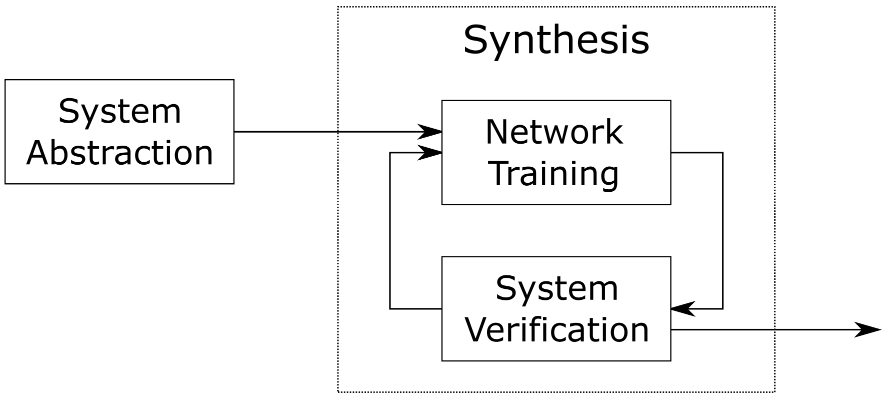

Figure 3-1: A schematic overview of the three different routines in the correct-by-design neural network synthesis scheme.

controllers are discussed. In addition to this, a distinction will be made between different methods to encode the controller's output, namely labelled and unlabelled neurons.

As discussed in Section 2-1-1, there is a large host of neural network topologies available in the field of deep learning. The most fundamental of which is arguable the multilayer perceptron, consisting of a number of fully connected layers of neurons. All the connections in this type of network are in a feedforward configuration. This has the advantage of being both conceptually simple and straight forward to train since it does not require the computationally more expensive backwards propagation through time algorithm. In this thesis work, the main focus will be on multilayer perceptron neural network controller, but it should be noted that the methodology is robust with respect to other neural network topologies.

For the neural network to function appropriately in a feedback configuration, the neural network needs to receive information on the state of the plant and transform this into an appropriate input for the plant. Throughout this thesis work, the multilayer perceptron neural networks will be assumed to use ReLU activation functions. Because of these activation functions, the neural network, as per Definition 2-1-1, will map according to:

$$f \; : \; \mathbb{R}_+^n \rightarrow \mathbb{R}_+^k \tag{3-1}$$

Since the state space is defined as $X \subseteq \mathbb{R}^n$ and the input space as $U \subseteq \mathbb{R}^m$ a normalization function and denormalization function is needed to map these spaces to $\mathbb{R}_+$ and back. The normalization function maps:

$$n \; : \; \mathbb{R}^n \rightarrow [0,1]^n \tag{3-2}$$

and the denormalization function maps:

$$n^{-1} \; : \; [0,1]^n \rightarrow \mathbb{R}^n \tag{3-3}$$

Since $[0,1]^n \subset \mathbb{R}_+^n$, a normalized representation can be used by a ReLU based neural network to represent the state of the plant. The neural network will thus receive a quantized and normalized representation of the state:

$$\vec{x}_{nn} = n(q(\vec{x})) \tag{3-4}$$

where $q(\vec{x})$ is the state quantizer as defined in Section 3-2.

Each input neurons of the neural network controller will then receive a single dimension of the quantized and normalized state vector $n(q(\vec{x}))$. Hence a four-dimensional plant would have a corresponding neural network controller with four input neurons. It should be noted that this particular method of encoding the state of plant scales well with plants of a higher dimensionality.

Equipped with a representation of the state of the plant, the neural network controller can now provide an input to the plant. One of the requirements for the employed reinforcement learning scheme, is that the output of the neural network (and thus the input to the plant) should be provided in a probabilistic manner. This leaves a few different approaches to encode the input of the plant in the neural network. Two of these approaches shall be discussed: labelled input neurons and unlabelled input neurons.

3-1-1 Labelled

In a labelled neuron topology, each and every input that belongs to the finite cardinality input space is assigned an individual output neuron. Hence, each and every output neuron is 'labelled' to represent an input in the finite cardinality input space. The value that each neuron outputs is the probability of that input being the appropriate input for that state. The sum of all the output neurons is therefore 1. A graphical representation of the described topology is depicted in Figure 3-2.

Mathematically the input to the plant given by the output of the labelled output neuron neural network controller topology is equal to:

$$\vec{y} = f(\vec{x}_{nn}, \boldsymbol{\theta}) = \begin{bmatrix} \Pr\{\vec{u} = \vec{u_1}\} \\ \Pr\{\vec{u} = \vec{u_2}\} \\ \ldots \\ \Pr\{\vec{u} = \vec{u_k}\} \end{bmatrix} \tag{3-5}$$

where f is the neural network as defined in Equation 3-1. Throughout this thesis work, the probabilistic inputs to the plant as a function of the neural network parameters $\boldsymbol{\theta}$ shall be referred to as $\vec{u}_p = f(\vec{x}_{nn}, \boldsymbol{\theta})$. In a probabilistic setting, the input to the plant is sampled and denoted as:

$$\vec{u} = \gamma(f(\vec{x}_{nn}, \boldsymbol{\theta})) \tag{3-6}$$

where $\gamma \ : \ [0,1]^k \to U_a$ is a sampling function. This sampling function samples the input based on the probabilities provided by the neural network. The set U_a is defined in Section 3-2. In a deterministic setting, the input to the plant is sampled greedily by taking the input with the highest probability as provided by the neural network. This is denoted throughout this thesis work as:

$$\vec{u} = \gamma_g(f(\vec{x}_{nn}, \boldsymbol{\theta})) \tag{3-7}$$

where $\gamma_g \ : \ [0,1]^k \to U_a$ is the greedy sampling function.

In order to illustrate the multilayer perceptron, normalized quantized state and labelled input neural network controller topology an example will be given.

Example 3-1-1. *Consider a three-dimensional plant with switched mode linear dynamics. The plant has* 5 *different modes and the objective of the neural network controller is to provide,*

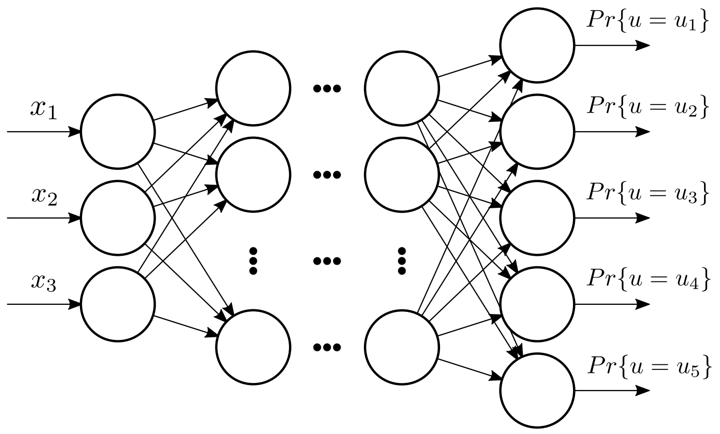

Figure 3-2: A graphical example representation of a multilayer perceptron neural network controller. This particular topology uses a normalized quantized representation the state of the plant as input to the neural network and labelled output neurons to represent the input to the plant.

based on the state of the system, in which mode the plant should operate such that a certain control specification is adhered. The input to the plant needs to be an integer number $u = \{1, 2, 3, 4, 5\}$ *that represents the mode in which the plant should be. In a labelled neuron topology, this calls for* 5 *output neurons which label to the respective mode of the plant. The required amount of input neurons is* 3 *in order to represent every dimension in the state space.*

The number of hidden layers that the neural network controller has is up to the user and based on the complexity of the required controller behaviour. This is often not known a priori and some intuition and/or iteration may be required in order to find a balance between attaining a proper neural network controller and the amount of data required to store the neural network controller.

The labelled neuron encoding method has the advantage of allowing for the use of some well-studied loss functions, primarily the cross-entropy (logarithmic) loss function. This particular loss function can be used because of the fact that the outputs are probabilities of labelled plant's inputs and sum to 1. The cross-entropy function is well studied in the context of deep learning and has some desirable properties. This causes the policy gradients to, in general, be 'more directed' in case of erroneous inputs and this leads to fast synthesis times when using conventional neural network libraries.

The main disadvantage to using these types of labelled neurons is that the amount of output neurons of the neural network controller is equivalent to the cardinality of the input space. Hence, for higher input space dimensions, the size of the neural network grows exponentially. Although the methodology is still valid for higher dimensionally input spaces, the neural network controller rapidly becomes unwieldy due to the large amount of output neurons required. This same phenomenon also severely slows down the neural network training speed when higher cardinality input spaces are used.

3-1-2 Unlabelled

In an unlabelled topology, the output neurons of the neural network controller do not map to all of the inputs in the finite input space. Instead the neurons are used to imply the appropriate input whilst still retaining the probabilistic requirement of the neural network's output. There are multiple ways to imply the input in a probabilistic manner. The method that will be considered in this thesis work achieves this by using two neurons per input space dimension. These two neurons imply the range in which the probabilistic inputs will be. The probability of each of the inputs that exist within this range is uniformly distributed over the range. The input selector then selects an input from this range based on this probability. As the range grows narrower, the neural network expresses it certainty of that input being the appropriate one by increasing the chance of it being picked. This is due to the fact that the probability of an input being used is inversely proportional to the width of the range of the output neurons. A graphical representation of the described topology is depicted in Figure 3-3.

Mathematically the input to the plant given by the output neurons of the unlabelled neural network controller topology is described by:

$$\vec{y} = f(\vec{x}_{nn}, \boldsymbol{\theta}) = \begin{bmatrix} y_1 & y_2 & \dots & y_{k-1} & y_k \end{bmatrix}^T \tag{3-8}$$

which implies the set of stochastic inputs $U_s \subseteq U_a$:

$$U_s = \{\vec{u}_s \in U_a \mid n_i^{-1}(y_{2(i-1)+1}) \leq u_s^i \leq n_i^{-1}(y_{2(i-1)+2})\} \tag{3-9}$$

where n_i^{-1} denotes the i'th dimension of the denormalization function of the input space, u_s^i is a value of the vector $\vec{u}_s$ at index i and f is the neural network as defined in Equation 3-1. The definition of the set U_a is provided in Section 3-2. In a probabilistic setting, the input to the plant is sampled from the set U_s such that $\vec{u} = \gamma(U_s)$ where $\gamma : U_s \to U_s$ is a sampling function. This sampling function samples an input $\vec{u}$ from the set of stochastic inputs U_s with probability $\Pr(\vec{u} = \vec{u}_s \in U_s) = \frac{1}{|U_s|}$. Since the set U_s is implied by the neural network output this will be denoted as $\vec{u} = \gamma(f(\vec{x}_{nn}, \boldsymbol{\theta}))$. In the deterministic setting, the input is defined as:

$$\vec{u} = q_u\left(\begin{bmatrix} n_1^{-1}(\frac{y_1+y_2}{2}) \\ .. \\ n_m^{-1}(\frac{y_{2m-1}+y_{2m}}{2}) \end{bmatrix}\right) \tag{3-10}$$

where $q_u(\vec{y})$ is the input space quantizer as defined in Section 3-2. To simplify the notation, this will similarly be referred to as $\vec{u} = \gamma_g(f(\vec{x}_{nn}, \boldsymbol{\theta}))$ with the greedy sampling function $\gamma_g : \mathbb{R}^k \to U_a$ mapping the implied input to an input in the set U_a. This notation will be used throughout the thesis work.

To illustrate the idea of a multilayer perceptron neural network controller with normalized quantized state and unlabelled inputs an example will now be given.

Example 3-1-2. *Consider a three-dimensional switched system with linear dynamics in every mode. The input that the plant requires is again an integer number that represents in which mode the plant should operate* $u = \{1, 2, 3, 4, 5\}$. *Similarly to the labelled case, the state of the plant will again be represented by three input neurons which receive a normalized quantized representation of the state of the plant. The input to the plant is represented in an unlabelled fashion by two normalized output neurons* u_{11} *and* u_{12}. *These neurons output a value between* $[0, 1]$. *The values of these two neurons are then denormalized to values between* $1 \leq u \leq 5$ *such that they span the set of stochastic inputs. During training these output neurons are*

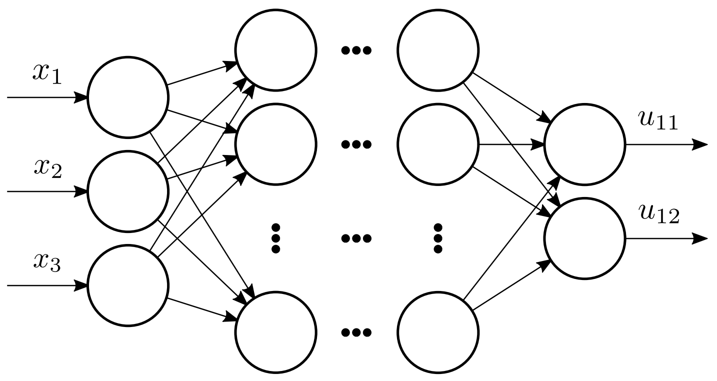

Figure 3-3: A graphical example representation of a multilayer perceptron neural network controller. This particular topology uses a normalized quantized representation the state of the plant as input to the neural network and unlabelled output neurons to represent the input to the plant.

probabilistically interpreted. For example, if the output is $u_{11} = 0.21$ and $u_{12} = 0.62$ this would imply the inputs $u = 1 \cup u = 2$ with probability $Pr\{u = 1\} = 0.5$, $Pr\{u = 2\} = 0.5$. If the neural network controller is evaluated deterministically, the average of the two output range neurons will be denormalized and quantized in order to find the input. Hence, if $u_{11} = 0.21$ and $u_{12} = 0.62$ then $\bar{u}_1 = 0.415$ which implies $u = 2$ when denormalized and quantized.

The main advantage to using this particular method of encoding is that, contrary to the labelled neuron topology, it scales well with input spaces that are of a higher cardinality. The disadvantage however is that it puts limitations on the loss functions and therefore policy gradients that can be used during reinforcement learning. In practice this can lead to long synthesize times. Furthermore, the topology also inherently restricts the selection of inputs to a subset of neighboring inputs. This also limits the neural network's ability to explore the input space during training.

3-2 System abstraction framework

In order to synthesize correct-by-design neural network controllers, verification of said controllers is an absolute requirement. It is therefore paramount to verify the neural network that is trained to act as a controller. In this thesis work, fixed-point algorithm based verification methods are used in order to verify the behaviour of the neural network controller. This does however require that numerical methods can be applied to the system, which in turn requires the system to exhibit a finite cardinality state space. In order to achieve this, the system abstraction framework routine, which initializes the entire synthesize procedure, sets up a system abstraction framework such that the resulting abstractions are of finite cardinality.

In order to achieve this abstraction framework, two quantizers are added in the feedback loop: a state quantizer $q : \mathbb{R}^n \to \mathbb{R}^n$ and an input quantizer $q_u : \mathbb{R}^m \to \mathbb{R}^m$. The resulting scheme is schematically visualized in Figure 3-4. These quantizers are introduced in order to yield a framework in which the state and input space as experienced by the controller are of finite cardinality. In order to achieve this, the quantizers also bound the space to a subset of the entire space. Consider the state vector $\vec{x} = [x_1, x_2, ..., x_n]^T$, the quantized state vector $\vec{\hat{x}}$ is

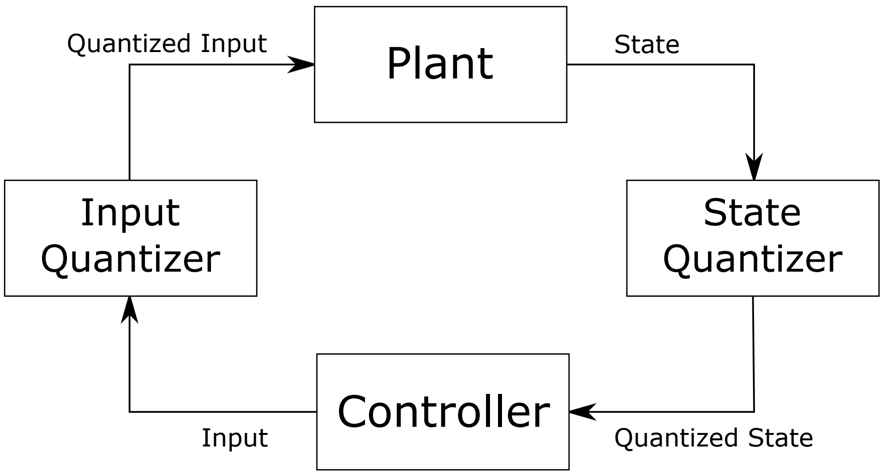

Figure 3-4: The control scheme that is used in order to achieve a finite cardinality abstraction framework as experienced by the controller.

defined element wise as:

$$\hat{x}_i = q(\vec{x})_i = \begin{cases} x_i^l & x_i < x_i^l \\ x_i^u & x_i > x_i^u \\ x_i^l + k\eta_i \; : \; |x_i - (x_i^l + k\eta_i)| < \eta_i \text{ with } k \in \mathbb{N}_0 & \text{otherwise} \end{cases} \tag{3-11}$$

where $\vec{x}^l$ and $\vec{x}^u$ are the lower and upper bounds of the state space respectively and $\vec{\eta}$ is the state quantization parameter. This particular quantization scheme ensures that there are only a finite number of points within the bounds of the space it quantizes. In this thesis work, the state quantizer that quantizes the state $\vec{x}$ is denoted as $\vec{\hat{x}} = q(\vec{x})$ with the quantization parameter $\vec{\eta}$. The input quantizer that quantizes the input $\vec{u}$ is denoted as $\vec{\hat{u}} = q_u(\vec{u})$ with the quantization parameter $\vec{\eta_u}$.

The resulting system, as experienced by the neural network controller, is equivalent to the original system with exception of the quantizers q, q_u and the sampling time τ. Consider the control system:

$$\Sigma = (X, U, \phi) \tag{3-12}$$

where $X \subseteq \mathbb{R}^n$ is the state space, $U \subseteq \mathbb{R}^m$ is the input space an ϕ the nominal dynamics of the plant. The resulting system abstraction framework system based on the system definition in Section 2-2-1 becomes:

- $X_a = \{\vec{x_a} \in \mathbb{R}^n \mid \exists \vec{x} \in X, \; \vec{x}_a = q(\vec{x})\}$
- $U_a = \{\vec{u_a} \in \mathbb{R}^m \mid \exists \vec{u} \in U, \; \vec{u}_a = q_u(\vec{u})\}$
- $x \xrightarrow[\tau\eta]{\tau} x'$ if there exists a solution $\xi_x : [0, \tau] \to \mathbb{R}^n$ of Σ satisfying $\|\xi_x(\tau) - x'\| \leq \eta$
- $Y_a = \mathbb{R}^n$
- $H_a = 1_X | X_a \to \mathbb{R}^n$

Using this scheme, it is possible to create an abstraction framework which has a state space X_a and input space U_a that is of finite cardinality and is equivalent to the original system with exception of the quantizers and sampling time. This property is thoroughly exploited during the verification phase, which is described in Section 3-4.

3-3 Network training

Once the abstraction framework is established the procedure switches over to the network training routine. The network training routine and the system verification routine are iterative routines which are repeated until the desired controller is synthesized. During the network training routine, the neural network controller is trained to function as a controller through the process of reinforcement learning on simulated episodes. During this training, information from previous verification routines is used to speed up the synthesize routine. This section will contain a detailed description of the network training routine.

3-3-1 Episodes

During the training routine, the neural network controller is trained using reinforcement learning, based on the performance of episodes. An episode is defined as a finite horizon iteration of the system, where the neural network is acting as a controller in closed loop as depicted in Figure 3-4.

During an episode, the current state of the simulated plant is fed through the state quantizer and into the neural network. The initial state of the plant x_i is chosen at the start of an episode from the training initial state set I_0. The quantized state of the plant $\vec{\hat{x}}$ is then normalized. The normalized quantized state of the plant is denoted as $\vec{x}_{nn}$. Every normalized quantized state dimension is then assigned to its corresponding input neuron of the neural network controller. This particular method of introducing the state to the neural network controller is independent of the neural network topology and type as detailed in Section 3-1. The neural network controller is then evaluated based on the normalized quantized state of the plant. Based on the mode of operation, the output of the controller is interpreted probabilistically or deterministically by the input selector. It is important that the neural network controller is evaluated probabilistically during the network training routine for reasons that will become apparent in Section 3-3-2. This probability represents the 'certainty' of the neural network, that the presented input is the appropriate one given the state of the plant. During training the input selector will sample and denormalize an input form the probabilistic inputs. That input is then fed to the input quantizer. The resulting quantized input is fed to the plant and the plant is iterated. This process repeats until a termination condition is met. The termination conditions are checked as soon as the quantized state is fed to the controller.

By repeating this process, a feedback controlled walk is generated in the state and input space and stored. These states and inputs can then be used by reinforcement learning techniques in order to train the neural network controller. A schematic representation of signal flow is represented in Figure 3-5.

Termination conditions

There are a number of different termination conditions that can be met in order to stop the episode. The most obvious one is simply hitting the finite horizon N. The finite horizon should be picked such that the specification can be adhered from every cell in the state space if possible. The exact finite horizon that is required in order to meet this particular requirement is often not known a priori and has a high dependency on the plant's dynamics

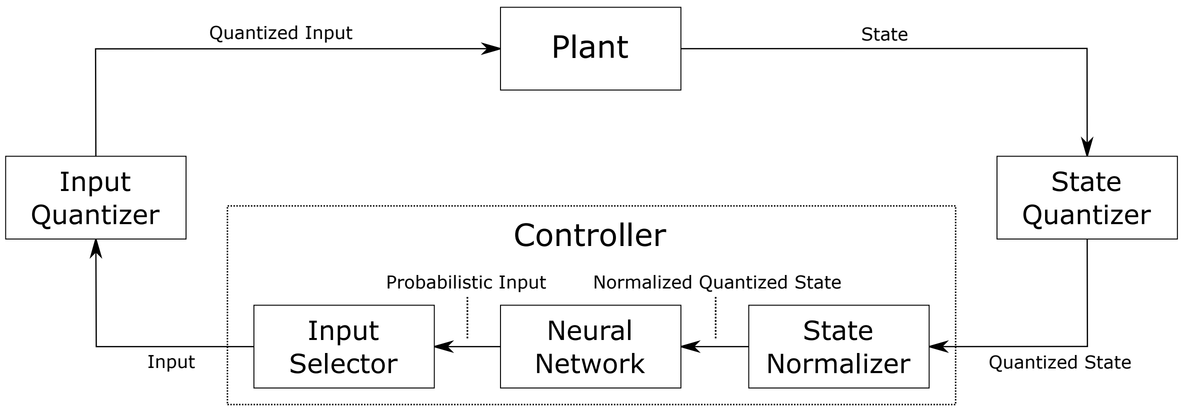

Figure 3-5: A schematic representation of the signal flow during a simulation of a finite horizon episode.

and the sampling time τ of the simulated plant. In practice some intuition and heuristics will thus be required in order to pick an appropriate finite horizon N.

Another obvious termination condition for an episode is leaving the confinement of the bounded state space in which the state quantizer operates. This termination condition is strictly required in order to guarantee the finite cardinality of the state space. Once the dynamics of the feedback controlled system venture outside of the bounds the episode is terminated.

Other than these more straightforward termination conditions, the episode is also subject to termination conditions based on the linear temporal logic control specification. For example, in case of a reachability specification, reaching the set for which reachability is in question is a sufficient termination condition. This condition is sufficient since the controller is only tasked with reaching the set. Therefore, once the set has been reached, the remainder of the episode is irrelevant for controller synthesize. Similarly, for an invariance specification, the episode is terminated as soon as the episode leaves the invariant set. For a reach and stay specification, the episode is terminated once the episode has entered the to be reached set and has then ventured out of it again.

Another optional termination condition is available in case of a reachability specification. In order to speed up the training process, it is also possible to terminate the episode once the previous winning set W is reached. The winning set W is formally defined in Section 3-4. Although using this method does speed up the training time, it does not have any formal guarantees since the winning set could have changed during the training process. This will further be elaborated upon in Section 3-4-3.

Plant iterations

As stated previously in this section, the simulated plant is iterated during the episode in order to observe the effect of the interaction between the neural network controller and the plant. During an episode, a numerical simulation is therefore used to approximate the actual plants behaviour. In order to describe the dynamics of the plant, ordinary differential equations are employed. These are then numerically integrated using the Runge-Kutta method for numerical integration. For $\dot{\vec{x}} = \phi(t, x, u)$ this yields:

$$\begin{aligned}
\vec{x}_{n+1} &= \vec{x}_n + \frac{1}{6}h\left(\vec{k_1} + 2\vec{k_2} + 2\vec{k_3} + \vec{k_4}\right)\\
t_{n+1} &= t_n + h\\
\vec{k_1} &= \phi(t_n, \vec{x}_n, u)\\
\vec{k_2} &= \phi(t_n + \frac{h}{2}, \vec{x}_n + h\frac{\vec{k_1}}{2}, u)\\
\vec{k_3} &= \phi(t_n + \frac{h}{2}, \vec{x}_n + h\frac{\vec{k_2}}{2}, u)\\
\vec{k_4} &= \phi(t_n + h, \vec{x}_n + h\vec{k_3}, u)
\end{aligned} \tag{3-13}$$

where h is the step size for the Runge-Kutta numerical integration. In practice this is simply an integer division of the sampling time τ. This provides a reasonably accurate approximation of the actual dynamics without losing too much of the computational power and synthesis time during the network training routine on iterating the plant.

3-3-2 Reinforcement learning

Based on the array of states and inputs that results from each episode, reinforcement learning is used to train the neural network controller. In reinforcement learning the neural network acts as an actor in the environment. By appropriately reinforcing and deterring the interactions that the neural network has with the environment it is possible to train the network to exhibit the desired control policy. A more thorough explanation of reinforcement learning is provided in Section 2-1-3.

Based on the performance of an individual episode, policy gradients are computed in order to reinforce or deter the control policy that the neural network controller exhibits. The performance of an episode is based on the specification that the neural network controller is being trained for. If the finite episode has adhered to the specification, the control policy will be reinforced. If it did not adhere to the specification the policy will be deterred.

Testing if an episode adhered to a reachability specification is done by checking if the final state of the episode is in the to be reached set. For other specifications such as invariance, checking whether or not the episode adheres to the specification is not trivial due to the fact that the episode is finite whereas the specification is not. Hence in case of invariance or reach and stay specifications a likelihood of adhering to the specification is the best result that can be achieved. Therefore in the case of an invariance specification, the control policy will be reinforced when the episode has terminated due to the finite horizon N and all of the states in the episode are in the invariant set. For a reach and stay specification, the control policy will be reinforced when all of the states of the episode are in the reach and stay set once that set has been entered and it has terminated due to the finite horizon N. In all other cases the neural network's control policy will be deterred.

If reachability is the specified control specification and more knowledge of the plant is available, norm-based reinforcement could also be employed. The idea is to compute the weighted norm of the initial state and compare it to the weighted norm of the final state at episode termination. By picking an appropriate norm and weights, a decrease in the norm could also be used as a performance criterion for reinforcing the control policy.

Once it is apparent whether or not the control policy needs to be reinforced or deterred, the policy gradients for the neural network can be calculated. There are various different methods available for calculating these policy gradients with different levels of complexity. In addition to this, how the policy gradients are calculated is also dependent on the neural network controller's topology and the loss function that is employed.

The simplest method to calculate these policy gradients is to reinforce or deter every state-input pair in the episode based on the performance of the episode. In order to illustrate an algorithm for finding these policy gradients, it will be assumed that the neural network controller uses labelled inputs (see Section 3-1-1) and is of a multilayer perceptron topology. In case of labelled inputs, reinforcing the state-input pair will generate a reinforcing input label for that state-input pair. This entails generating a label where the input that was used for that state has a certainty of 100% and all the other inputs have a certainty of 0%. This allows for the computation of the policy gradient that reinforce the control policy. A gradient descent step will then make the state-input pair more 'certain' and hence increase the change that the neural network controller will output that policy in the future. If the input for that state is to be deterred, for labelled inputs, a deterring label will be generated which will put all the inputs at 100% except for the input that was used. This algorithm is depicted in Algorithm 1. In this algorithm $\mathcal{P}$ is the performance flag of the episode which is 1 if it 'adheres' to the specification and 0 otherwise, $\mathcal{X}$ is the array of normalized quantized states $\vec{x}_{nn}$ in the episode and $\mathcal{U}$ is the array of inputs in the episode. The function $i : \mathbb{N}_0 \to U_a$ maps the index j of the output neurons to the actual input which that neuron labels.

The algorithm for calculating the policy gradient for the neural network controller as stated is just an example of such an algorithm. This algorithm can be extended to other neural network controller types and topologies, such as the unlabelled input neurons controller topology. Furthermore, more sophisticated algorithms could be developed that reinforce based on the contribution that each state-input pair has on the performance of the episode.

3-4 System verification

In Section 3-3 the employed methodology to synthesize neural network controllers is outlined. However, after training the resulting neural network controller still lack the formal guarantees that ensure the controlled system adheres to the predefined specification.

As discussed in Section 2-2, correct-by-design control often employs finite cardinality abstractions of the original system in order to synthesize controllers. The same techniques can also be used to verify the controller's behaviour. Once the neural network controller has been trained for an arbitrary number of episodes, the network training routine is switched over to the system verification routine. In the system verification routine, the abstracted feedback controlled system is considered and verified using fixed-point based verification algorithms. These numerical verification algorithms can be used due to the fact that the abstracted system is of finite cardinality. Any guarantees that the verification algorithm yields for the abstracted system also hold for the original system. This is due to the fact that these two systems are behaviourally equivalent due to the simulating relation between them (see Section 3-2).

Algorithm 1 Computation of the policy gradients and training of a neural network controller using reinforcement learning for labelled inputs.

```
1: procedure ReinforcementLearningController($\mathcal{P}$, $\mathcal{X}$, $\mathcal{U}$)
2:     $\mathcal{L} \leftarrow 0$
3:     for $i \leftarrow 1, N$ do
4:         $\vec{x}_{nn} \leftarrow \mathcal{X}_i$
5:         $\vec{u} \leftarrow \mathcal{U}_i$
6:         $\vec{u}^{nn} \leftarrow f(\vec{x}_{nn}, \boldsymbol{\theta})$
7:         for $j \leftarrow 1, |\vec{u}|$ do
8:             if $\mathcal{P} = 1$ then
9:                 if $\vec{u} = i(j)$ then
10:                    $L_j \leftarrow 1$
11:                else
12:                    $L_j \leftarrow 0$
13:                end if
14:            else
15:                if $\vec{u} = i(j)$ then
16:                    $L_j \leftarrow 0$
17:                else
18:                    $L_j \leftarrow 1$
19:                end if
20:            end if
21:        end for
22:        $\mathcal{L} \leftarrow \mathcal{L} - \sum_{j=1}^{|\vec{u}|} \|L_j\| \log(u_j^{nn})$
23:    end for
24:    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \lambda \nabla_\theta \mathcal{L}$
25: end procedure
```

In this section the verification method that is employed to verify the controlled system's behaviour is outlined. First the notion of a partial abstraction will be described. Then the use of fixed-point algorithms shall be explained in order to find the set for which the controller adheres to the specification. Finally, the integration between the system verification routine and the neural network training routine shall be discussed.

3-4-1 Partial abstraction

Once the synthesize procedure enters into the system verification routine, a partial abstraction of the controlled system is created. The abstraction is only a partial one since the transitions that the system is allowed to take are constraint by the neural network controller. The partial abstraction of the system is a system that simulates the origin system via the feedback refinement relation. This particular simulation relation is reviewed in Section 2-2-4.

The main idea is to create a set of hyper-rectangular cells (hyper-cells) that cover the bounded state space as imposed by the state space quantizer. Each of these cells is then a subset of the state space and the behaviour of these cells is considered instead of every point in the bounded state space. This ensures that the resulting partial abstraction is of a finite cardinality and therefore numerical methods can be utilized. By employing this simulating abstraction, the behaviour of a state in the bounded state space can be over-approximated by considering the hyper-cell in which that state is included. The behaviour of the hyper-cell can then be used to provide guarantees on the behaviour of the state given that the plant is being controlled by the neural network controller.

Mathematically the partial abstraction system S_p that simulates S via the feedback refinement relations can now be described:

- $\bar{X}_p$ is a cover of the state space X by non-empty, closed hyper-intervals and every element $x_p \in \bar{X}_p$ is compact.
- $U_p \subseteq U_a \mid \forall u_p \in U_p,\ \exists \vec{c} \in \bar{X}_p$ s.t. $\gamma_g(f(\vec{c}, \boldsymbol{\theta})) = u_p$
- $(x_p, u_p, x_p') \in \rightarrow\ \mid x_p \in \bar{X}_p,\ u_p = \gamma_g(f(\vec{c}, \boldsymbol{\theta})) \in U_p$ and $x_p' \in \bar{X}_p$ s.t. $\mathrm{Post}_{u_p}(x_p) \cap x_p' \neq \emptyset$
- $\mathrm{Post}_{u_p}(x_p) = \emptyset$ whenever $x_p \in X \setminus \bar{X}_p, u_p \in U_p$

In this definition $\gamma_g(f(\vec{c}, \boldsymbol{\theta}))$ denotes the greedy input from the neural network controller, $\vec{c}$ is the center of the hyper-cell and $\mathrm{Post}_{u_p}(x_p)$ is a function that returns the set to which the dynamics of the plant map based on the initial cell and the input. In actuality, the neural network is fed a normalized quantized representation of the cell centers $f(n(q(\vec{c})), \boldsymbol{\theta})$. For readability, the normalization and quantization functions are ommited.

Once the partial abstraction is formally established the over-approximation function is required in order to compute that transitions that the partial abstraction makes based on the neural network controller's input.

To find the input as prescribed by the neural network controller for a given hyper-cell, the center of that cell is presented to the neural network controller. The output of the neural network controller is then greedily chosen to be the output with the highest probability. Based on that, the set of hyper-cells that over-approximate the dynamics can be computed. The over-approximation function thus provides a mapping from single hyper-cells to a set of hyper-cells. In order to save computational power and time during synthesis, it is possible to save the transitions that have already been computed during synthesis. This slowly generates a fuller abstraction until at some point the neural network has exhausted all possible transitions. This method however scales poorly with higher state and input spaces, therefore it is also possible to compute the transitions of the system ad hoc and not store them. This frees up memory and allows for larger state and input spaces at the cost of longer synthesis times.

How the over-approximation function is defined is based on the dynamics of the plant. For linear dynamics less conservative over-approximations can be computed that for nonlinear dynamics. Both means of defining the over-approximation function will now be discussed.

Linear dynamics

For linear dynamics, the linearity of the plant can be exploited to compute a less conservative over-approximation than is available for non-linear dynamics. The important observation here is that iterating the vertices of the hyper-cell with the plant's dynamics would result in a convex hull that describes the set of states to which any state in the initial hyper-cell maps. The set of hyper-cells that defines the over-approximated set therefore consists of all the hyper-cells that intersect with this convex hull.

Finding the hyper-cells that intersect with the convex hull of the dynamics is not a trivial task. In this thesis, this is achieved by performing a flood fill algorithm that starts at the center of the resulting convex hull. The hyper-cell that includes that center is by default added to the set with which the convex hull intersects. From that cell, the flood fill algorithm procedurally adds the neighboring cells to be checked for intersection as long as those cells have not already been checked. In this manner every cell that could potentially intersect with the convex hull is checked. Individual hyper-cells are checked for intersection by checking if at least one of the vertices of that hyper-cell is within the bounds of the convex hull. This is done by finding the linear functions and normals that describe each of the hyper-planes of the convex hull. The normals are calculated such that the center of the convex hull is considered inside of the convex hull. Checking each if a point is in the convex hull is then simply a matter of checking whether or not the projection onto the normal of the hyper-planes is positive or negative. If all projections are positive it means that the point is in the convex hull. If anyone of the projections is negative it means it is outside of the convex hull. Thus, for every hyper-cell that the flood fill algorithm processes, it checks if the vertices of the hyper-cell under investigate are inside of the convex hull. If any of the vertices are inside the convex hull, it means the hyper-cell intersects the convex hull and is thus added to the new state set X'. This flood fill algorithm automatically terminates once all the intersecting cells have been checked and added to the new state set X'. The over-approximation function returns the new state set X'.

The algorithm for finding the intersecting hyper-cells with the plant iterated convex hull is described in Algorithm 2. In this algorithm $\mathcal{N}$ represents the set of normal vectors that are normal to the edges of the convex hull and point to the center of the convex hull, $\mathcal{V}$ are the vertices of the convex hull, c is the center of the convex hull and d the distance metric:

$$d \; : \; \bar{X}_p \times \bar{X}_p \to \mathbb{R} \tag{3-14}$$

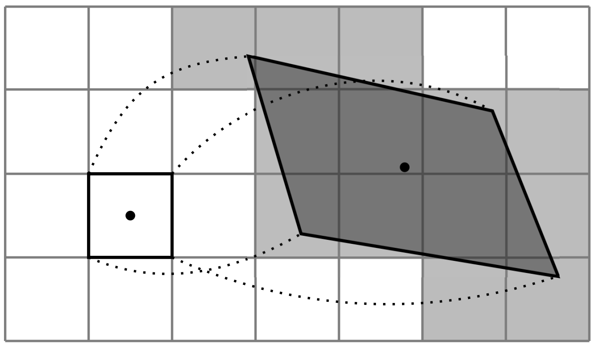

Figure 3-6: An example of the over-approximation for a two dimensional example with linear dynamics. Given a single hyper-cell (the one on the left) the plant's dynamics would result in the transitions to the set of hyper-cells, indicated in light gray, based on the convex hull of the iterated single hyper-cell, indicated in dark gray. The resulting set is a simulating over-approximation of the linear dynamics of the actual plant.

that maps two hyper-cells in $\bar{X}_p$ to a distance metric. The observant reader will note that this algorithm only works for spaces with a dimension of at least 2. For 1 dimensional systems this algorithm is adjusted to check whether or not the vertices are inside or outside of the bounding vertices.

In order to illustrate the idea, Figure 3-6 represents the over-approximated set for a single hyper-cell. In the figure the plant is two dimensional and thus the state space is also two dimensional. The light gray cells represent the set of hyper cells to which the origin hyper-cell maps. The dark gray shape is the convex hull to which any state in the initial cell maps based on the dynamics of the plant.

To illustrate why this particular method is less conservative than the traditional method for linear systems, an illustration demonstrating the traditional method is provided in Figure 3-7. The traditional method here refers to the method as employed by PESSOA[23] and SCOTS[31] (if the growth bound exactly encapsulates the outer vertices) for linear systems. Comparing Figure 3-6 and Figure 3-7 indeed reveals that the over-approximation based on the convex hull is considerably less conservative. It should however be noted that it is computationally significantly more expensive due to the flood fill algorithm when compared to the traditional method.

Nonlinear dynamics

For nonlinear dynamics, the convex hull methodology as discussed in the previous subsection is not available since the plant does not exhibit linear dynamics. Hence a different approach to finding the over-approximation function needs to be employed. In this thesis work, for nonlinear dynamics, the same methodology is employed as used in the synthesis tool SCOTS by Rungger et al. [31]. They propose the use of a radial growth bound function $\beta(r, \vec{u})$ which provides a radial bound on the dynamics where r is the radius of the hyper-cell and $\vec{u}$ the input for that cell. The input for a particular hyper-cell in the state space is found, similarly to how it is found for linear dynamics, by providing the neural network controller with the normalized quantized center of the hyper-cell and greedily picking the input from the probabilistic output of the network.

Algorithm 2 Flood fill algorithm to find the hyper-cells intersecting the convex hull

```
procedure FindIntersections($\mathcal{N}$, $\mathcal{V}$, $\vec{c}$)
    $X' \leftarrow \{\}$                                   ▷ Hyper-cells that intersect with the convex hull
    $I \leftarrow \{\vec{c}\}$                             ▷ Unprocessed hyper-cells
    $P \leftarrow \{\}$                                    ▷ Processed hyper-cells
    while $|I| > 0$ do
        $h \leftarrow I(0)$                                ▷ Get first hyper-cell in unprocessed hyper-cells
        $V \leftarrow$ Vertices of hyper-cell $h$
        $i \leftarrow 0$                                   ▷ Hyper-cell intersects flag
        for all $\vec{v} \in V$ do
            $f \leftarrow 1$                               ▷ Vertex in convex hull flag
            for all $n \in \mathcal{N}$ do
                $\vec{n_v} \leftarrow \mathcal{V}_n$       ▷ Vertex of convex hull that belongs to that normal
                $\vec{\delta} \leftarrow (\vec{v} - \vec{n_v})$
                $p \leftarrow \vec{n} \cdot \vec{\delta}$  ▷ Project vertex onto edge
                if $p < 0$ then
                    $f \leftarrow 0$
                    break
                end if
            end for
            if $f = 1$ then
                $i \leftarrow 1$
                break
            end if
        end for
        if $i = 1 \cup (\vec{c} \cap h \neq \emptyset)$ then
            $X' \leftarrow X' \cup h$                      ▷ Add hyper-cell to set of transitionable hyper-cells
            $I \leftarrow I \ \cup \ \{x \in X \setminus P \mid d(h, x) \leq \eta\}$ ▷ Add unprocessed neighboring hyper-cells
        end if
        $I \leftarrow I \setminus h$                       ▷ Remove hyper-cell from unprocessed cells
        $P \leftarrow P \cup h$                            ▷ Add hyper-cell to processed cells
    end while
end procedure
```

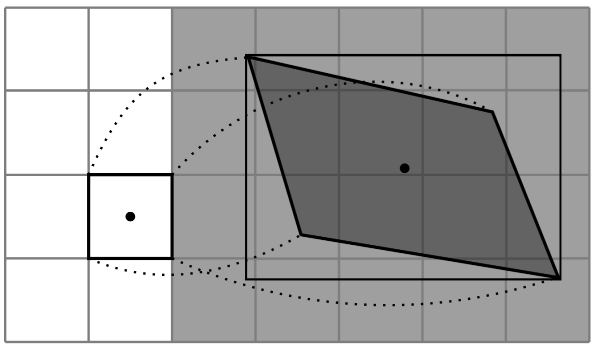

Figure 3-7: An example of a more conservative over-approximation for a two dimensional example with linear dynamics. Given a single hyper-cell (the one on the left) the plant's dynamics would result in the transitions to the set of hyper-cells, indicated in light gray, based on the intersection with the rectangle that encapsulates the outer vertices.

In their paper on the feedback refinement relation, Reissig et al. [28] also provide a general method to compute the radial growth bound based on the Jacobian of the plants dynamics. The Jacobian here is defined as the matrix L with:

$$L_{i,j}(u) \geq \begin{cases} D_j f_i(x,u) & \text{if } i=j \\ |D_j f_i(x,u)| & \text{otherwise} \end{cases} \tag{3-15}$$

Based on this definition of the Jacobian the radial growth bound for nonlinear plants becomes:

$$\beta(r,u) = e^{L(u)\tau} r + \int_0^\tau e^{L(u)s} w \,\mathrm{d}s \tag{3-16}$$

where τ is the sampling time of the plant, r is the radius of the hyper-cells and w is a disturbance term. The radial growth bound thus becomes $r' = \beta(r, \vec{u})$. The set of hyper-cells to which a hyper-cell can transition X' and which describes the over-approximation function then comes:

$$X' = \{x_p \in \bar{X}_p \mid (\phi(\tau, \vec{c}, f(\vec{c}, \boldsymbol{\theta})) + [\![-r', r']\!]) \cap x_p \neq \emptyset\} \tag{3-17}$$

where $\phi(\tau, \vec{c}, \vec{u})$ describes the nominal dynamics of the plant.

The idea of using a radial growth bound to compute the transfer function is illustrated for a two dimensional example in Figure 3-8. The light gray hyper-cells represent the set of hyper-cells with which the growth bound intersect and hence form the set to which the initial hyper-cell on the left maps. The dark gray hull represents the non-convex hull to with any state in the initial hyper cell maps based on the dynamics of the nonlinear plant.

3-4-2 Fixed-point algorithms

Based on the partial abstraction, fixed-point algorithms can now be used to find the subset of the finite cardinality state space for which the neural network controller adheres to the predefined control specification. The fixed-point algorithms, that are used for this purpose, are based on the control specification to which the controller needs to adhere. A more thorough review of fixed-point algorithms and there use is provided in Section 2-2-5.

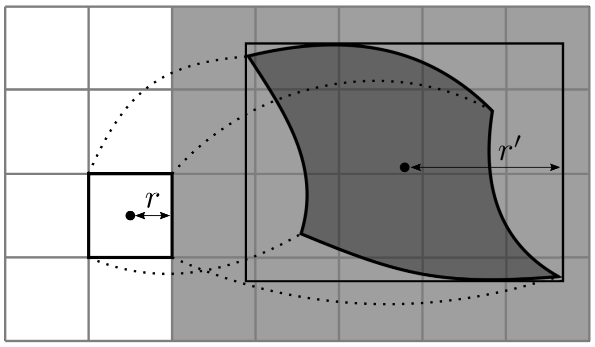

Figure 3-8: An example of an over-approximation for a two dimensional example with nonlinear dynamics. Given a single hyper-cell (the one on the left) the plant's dynamics would result in the transitions to the set of hyper-cells indicated by the light gray colour. This set is the set of hyper-cells that intersect with the radial growth bound. The dark gray non-convex hull is a simulating over-approximation of the nonlinear dynamics of the actual plant.

For a reachability control specification, the fixed-point algorithm needs to find the set for which the feedback controlled system is able to reach the to be reached set. Given the partial abstraction as discussed in the previous section, finding this set is a matter of iterating the reachability fixed-point operator. The fixed-point operator for a reachability specification is defined in Definition 2-2-11. For the partial abstraction, where the inputs are fixed to those of the neural network controller, this operator is defined as:

$$G_W(W) = \{x_p \in X_p \mid x_p \in Z \,\cup\, u_p = \gamma_g(f(\vec{x}_{nn}, \boldsymbol{\theta})),\ \emptyset \neq \mathrm{Post}_{u_p}(x_p) \subseteq W\} \tag{3-18}$$

where W is the resulting winning set and Z is the to be reached set. Iterating this set operator is guaranteed to end in a fixed-point. Reaching such a fixed-point will reveal the conservative set for which the neural network controller is able to adhere to the predefined specification.

For an invariance control specification, the same strategy can be applied. The fixed-point operator for an invariance specification is defined in Definition 2-2-10. For the partial abstraction, the fixed-point operator becomes:

$$F_W(W) = \{x_p \in W \mid x_p \in Z \,\cap\, u_p = \gamma_g(f(\vec{x}_{nn}, \boldsymbol{\theta})),\ \emptyset \neq \mathrm{Post}_{u_p}(x_p) \subseteq W\} \tag{3-19}$$

where W is the resulting winning set and Z is the invariant set. Similarly to the reachability operator, iterating this set operator is guaranteed to end in a fixed point. The resulting set W is the set for which the neural network controller is guaranteed to keep the plant in the invariant set Z.

This strategy can be extended to combinations of these fundamental control specification. An example of which would be the reach and stay control specification as defined in Definition 2-2-12. This would be achieved by first finding the invariant set and then applying the reachability operator to find the set for which that invariant set is reachable.

Performing such fixed-point iterations using partial abstractions thus provides formal guarantees on the behaviour of neural network feedback controlled systems. Since the partial abstractions simulate the original systems via the feedback refinement relation, the resulting guarantees also hold for the original system. It can thus be said that the synthesis procedure produces correct-by-design neural network controllers with the required guarantees on their behaviour.

3-4-3 Synthesis guiding

The network training and system verification routine, as discussed in the previous sections, result in correct-by-design neural network controllers with the required guarantees. The network training and system verification routines can provide this independently of one another, meaning that no interaction between them is required. It should however be noted that the system verification routine provides meaningful insights on the subset of the state space for which the neural network controller works appropriately and for which it does not. This information can be used to guide the neural network training routine and hence make it more expedient. There are a number of options available to achieve this. In this section a number of these options shall be discussed. Throughout this section, the set for which the neural network controller is able to adhere to the predefined control specification, is called the winning set $W \subseteq X_p$.

Perhaps the most intuitive method to integrate the verification and the synthesize routine is to change the initial state set I_0. The initial state set I_0 provides a set of states from which the network training routine picks a state at the start of each new episode. By changing this initial state set I_0 based on the winning set W, it is possible to let the network training routine 'focus' on a certain subset of states for which the controller currently does not adhere to the specification. In practice it is more useful to provide the network training routine with a few of these initial state focuses and switch between them in a round-robin fashion. This makes sure the neural network does not focus excessively on one particular subset, which could cause the network to 'lose' its broader and perhaps already correct behaviour.

To illustrate the idea of training focuses during synthesis, a number of training focuses shall now be described.

- All states: The initial state set I_0 contains all states in the state space in case of reachability and the entire invariant set in the case of invariance. This is the most generic and default training focus.
- Single state: The initial state set I_0 is restricted to a single state $I = \{s\}$, this causes the network training routine to repeatedly train on a single state and exhaustively explore all the control possibilities.
- Radial outwards: The initial state set I_0 is progressively grown radially from the goal state set. This training focus is of particular use in a reachability control specification. By progressively and radially expanding the initial state set around the goal set, the neural network controller first learns to control the system when it is close to the goal set. By radially expanding the initial state set I_0 the neural network controller can then 'build' on previously attained knowledge to expand its winning set W.
- Losing states: The complement of the winning set W^c is used as the initial state set I_0. This causes the training to focus on all the states for which the neural network controller currently cannot adhere to the specification. The advantage of using this focus is that is gives the controller ample opportunity to learn to control the system from initial states that are currently not in the winning set. The disadvantage is that these states could potentially be uncontrollable simply due to the nature of the plant. This would cause the network to try to learn an uncontrollable state and thus waste computational power and synthesis time.

- Neighbouring losing states: Similarly to the losing states training focus, this training focus focuses on states that are in the complement of the winning state W^c. The difference is that it only considers losing states that neighbour states in the winning set W. The idea is, similarly to the radial outwards training focus, that the neural network controller can then easily learn the required behaviour to also include this currently losing state.

It should be noted that this list of possible training focuses is not exhaustive and more and novel ways of integrating the system verification routine and network training routine using these focuses could be explored in the future.

Another method of integrating the system verification and network training routine is to reinforce the control policy whenever an episode enters into the winning set W in case of a reachability specification. This means that entering the winning set W will become an additional episode terminating condition and that ending an episode inside the winning set W will reinforce the control policy. The idea is that once the episode enters the winning set, that a previous system verification routine call has already determined, the controller already adheres to the specification. It can hence reinforce the control policy as from that state it can reach the goal set. It should be noted that during the network training routine, it is no longer certain whether or not a hyper-cell in the winning set still is contained within the winning set, due to the fact that the control policy is being adjusted. The hope is that the neural network will retain most if not some of its previously learned behaviour and learn to steer the state to a state from which it can already control the system appropriately.

3-5 COSYNNC

The methodology as discussed in this chapter has been implemented into a software correct-by-design neural network synthesis framework called COSYNNC[2]. The software is intended as a framework which can be extended by the user in order to generate correct-by-design neural network controllers according to the user's specification. These specifications include the neural network controller topology, neural network topology, plant dynamics and control specification. The framework was written in a C++14 environment and is currently limited to single thread CPU based execution. The framework uses the MXNet deep learning library[3] for all of the native neural network manipulations. An overview of the structure of the framework is depicted in Figure 3-9.

In its most primitive form, a correct-by-design neural network controller can be synthesized using COSYNNC by using the native multilayer perceptron neural network controller topology as provided by COSYNNC. The user is then required to provide the appropriate plant dynamics in the form of a class that inherits from the generic *Plant* class. For linear dynamics this would simply comprise of the dynamics of the plant. For nonlinear dynamics this would comprise of the dynamics of the plant and a radial growth bound on the dynamics of the plant. Once the plant is specified the user specifies the state and input quantizer parameters, which implicitly define the partial abstractions. The user is then required to define

[2]The framework is available at: https://github.com/WardvanderVelden/COSYNNC
[3]The MXNet deep learning library can be found at: https://mxnet.apache.org/

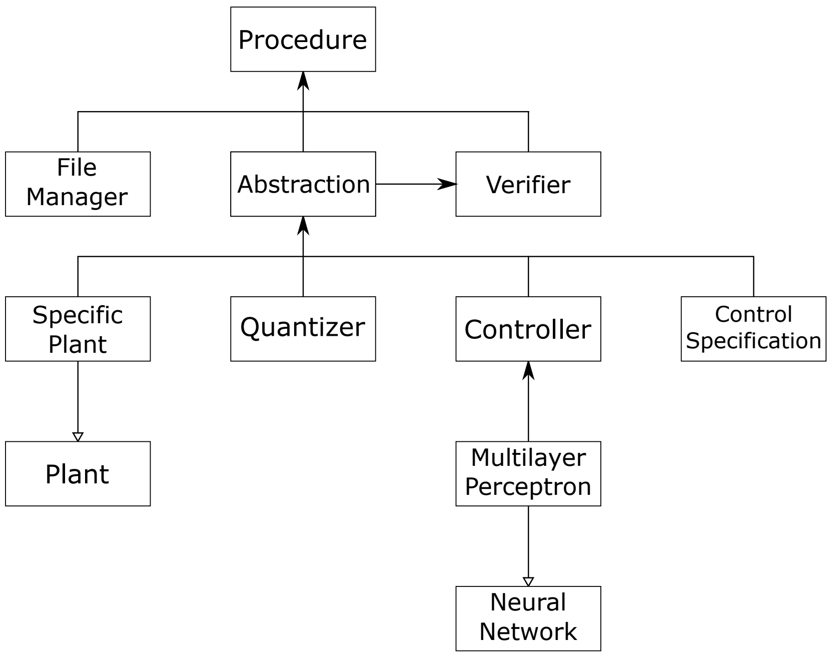

Figure 3-9: An overview of the structure of the COSYNNC framework. The filled arrows indicate that the class is instantiated or used by the class towards which it points. The hollow arrows mean that the class from whence that arrow came inherits its properties and behaviour from the class towards which the arrow points.

the synthesis parameters such as the episode finite horizon N, the control specification and the different training focuses that should be used during training. In addition to these fundamental synthesis parameters, there are also some additional and experimental parameters and functionalities that can be used such as norm-based reinforcement (reinforcing when the norm of the episode decreases) and winning set reinforcement (reinforcing when the episode ends in a previously calculated winning set). An example of the code required to synthesize a reachability controller for the Rocket example (as discussed in Section 4-2-1) is provided in Appendix A.

After these parameters have been defined and the framework has been successfully built, the synthesis procedure can be executed. During the synthesis procedure, the methodology as described in this chapter will be executed in code in order to synthesize a correct-by-design neural network controller. During each system verification routine, the framework will save the current neural network controller. In addition to this it is also possible to save the winning set, a SCOTS like static controller and the transitions that the partial abstraction contains during the system verification routine. The percentage size of the winning set W that the verifier found, with respect to the size of finite cardinality state space set, will be logged during synthesis. An overview of all the different outputs that COSYNNC can natively provide and their respective definition is given in Table 3-1.

Besides the ability to synthesize correct-by-design neural network controllers, COSYNNC also has an encoder which allows encoding the winning set W of the correct-by-design neural network controller as a neural network. The neural network controller itself and a neural network encoded representation of this set together form a full correct-by-design controller.

Table 3-1: An overview of the different outputs that COSYNNC can natively provide.

Name	Definition	Extension
Raw Neural Network	A minimalistic representation of the topology, the weights and biases are encoded as doubles and stored hexadecimally	.raw
MATLAB Neural Network	A representation of the weights and biases using matrices and vectors such that it can be read out and used in a MATLAB environment	.m
MATLAB Winning Set	A representation of the winning set such that it can be read out and used in a MATLAB environment	.m
MATLAB Controller	A representation of the resulting controller as a look-up table such that it can be read out and used in a MATLAB environment	.m
Static Controller	A representation of the resulting controller as a look-up table formatted as readable by SCOTS	.scs

This allows for the creation of a full correct-by-design control structure which boasts the same properties as a traditional correct-by-design controller.

As part of this thesis work, some additional tools were developed to complement and compare the resulting correct-by-design neural network controllers. One of these tools allows for the conversion of the partial abstraction transition file, as presented by COSYNNC, into a traditional nondeterministic correct-by-design controller using SCOTS. This allows for comparison between the COSYNNC verified controller and a traditional SCOTS based controller. In addition to this, another tool was developed which converts the correct-by-design neural network controller into a binary decision diagram that represents the same information. This also serves as a metric for comparison between neural network controllers and traditional binary decision diagram stored controllers.

Chapter 4

Results

In this chapter results will be presented that validate the methodology to synthesize correct-by-design neural network controllers as described in Chapter 3. In order to do so, first the metrics which are employed in order to compare the different correct-by-design controllers are discussed. After that, a few different plants will be considered and trained using different training parameters and compared using these metrics.

The considered classes of systems are: linear systems, linear hybrid systems, nonlinear systems and nonlinear hybrid systems. With linear and nonlinear systems, the notion of systems that have only a single mode with linear or nonlinear dynamics is meant. With linear and nonlinear hybrid systems, the notion of systems with multiple modes with linear or nonlinear dynamics is meant.

4-1 Metrics

In order to appropriately present the results and derive a meaningful conclusion from those results, it is paramount to define a few metrics and their meaning. These metrics will be evaluated for the plants presented in the results. The primary, non-quantifiable metrics are, a visual inspection of the controller's behaviour and its winning set W and a visual inspection of the partitions as presented by the neural network controller. This visual inspection allows one to confirm that the controller is exhibiting its intended behaviour and hence acting appropriately.

The quantifiable metrics that are presented in this chapter will primarily consist of data sizes in bytes that represent the number of bytes required to store a certain type of controller. These metrics will include the data requirement of the neural network controller in bytes (NN), the data requirement of a binary decision diagram encoding the winning set W in bytes (W BDD) and a binary decision diagram encoding the nondeterministic SCOTS correct-by-design controller using the same abstraction (BDD).

It should be noted that the sum of the data required for the correct-by-design neural network controller (NN) and a binary decision diagram encoding the winning set W (W BDD) encodes the same information as a correct-by-design controller traditionally would. Hence the sum of the two form another metric of interest (NN + W BDD).

In addition to these metrics, a binary decision diagram stored representation of the neural network controller (NN BDD) is also computed. This stores the same information as the neural network (NN) and binary decision diagram of the winning set W (W BDD). It therefore provides an insight into the compressive capability of both binary decision diagrams and neural networks. Furthermore, it is also a deterministic correct-by-design controller just like the correct-by-design neural network controller. It will therefore be taken as the baseline for comparing the data requirements for the correct-by-design neural network controllers to the traditional correct-by-design controllers.

There are also more refined methods of determinizing nondeterministic correct-by-design controllers. These methods attempt to pick inputs such that the binary decision diagram that encodes the controller is minimized. In this thesis work, the methodology and tools as presented by Zapreev [37] will be used to determinize the nondeterministic correct-by-design controllers as synthesized using SCOTS and stored as binary decision diagrams (D BDD). Unfortunately, these tools are only available for a version of SCOTS that was not utilized in this thesis work. Therefore, this D BDD comparison is not available for all of the plants.

Since the binary decision diagram of the winning set W (W BDD) still suffers the same issues as an ordinary binary decision diagram, a neural network representation of the winning set is also synthesized (W NN). This neural network has two output neurons that return an output larger than a predefined threshold σ based on whether or not the state of the plant is in the winning set or not. Combined with the correct-by-design neural network controller (NN + W NN), these two neural networks encode the same information as a traditional correct-by-design controller.

A concise overview of all of these different data requirement metrics for the different structures and their abbreviations is presented in Table 4-1.

Another relevant metric is the amount of time that is required to synthesize these different structures. The main structures that are of interests with respect to the required synthesis time are the correct-by-design neural networks themselves (NNt), the neural network encoding the winning set (WNNt) and the traditional correct-by-design controllers stored as binary decision diagrams (BDDt). The reason why only these structures are deemed to be interesting is because these are the only structures that are actually synthesized. The other structures are created by converting one data structure to another data structure which takes, relatively speaking, only a trivial amount of time. The additional time it takes is in the order of 1 second. Hence the time required for the other structures is simply the time required to synthesize the base controller plus a small bias. This implies that for both the W BDD and the NN BDD the synthesis time is equivalent to NNt and for the D BDD it is equivalent to BDDt. A table that depicts these time metrics and their abbreviations is presented by Table 4-2. All of the structures of interest are generated on CPU based software running on an i5-7500 3.4 GHz CPU overclocking to 3.8 GHz and 16 GB of RAM at 2400 MHz unless noted otherwise.

The final metrics have to do with the completeness of the different neural network structures. One of these metrics is the completeness k of the correct-by-design neural network controller with respect to a traditional correct-by-design controller. The completeness is defined as the size of the winning set W_s for the controller synthesized using SCOTS and the size of the

Table 4-1: The abbreviation and meaning of the data requirement metrics

Abbreviation	Unit	Meaning
NN	#bytes	Data requirement of the neural network controller
W BDD	#bytes	Data requirement of a binary decision diagram encoding the winning set W
NN + W BDD	#bytes	The combined data requirement of NN and W BDD
W NN	#bytes	Data requirement of a neural network encoding the winning set W
NN + W NN	#bytes	The combined data requirement of NN and W NN
NN BDD	#bytes	Data requirement of a binary decision diagram of the neural network controller encoding only the inputs for states inside the winning set W
D BDD	#bytes	Data requirement of a binary decision diagram encoding a deterministic SCOTS correct-by-design controller
BDD	#bytes	Data requirement of a binary decision diagram encoding the nondeterministic SCOTS correct-by-design controller

Table 4-2: The abbreviation and meaning of the time metrics

Abbreviation	Unit	Meaning
NNt	s	The amount of time required to synthesize the correct-by-design neural network controller
WNNt	s	The amount of time required to train a neural network to encode the winning set W
BDDt	s	The amount of time required to synthesize the traditional correct-by-design controller using SCOTS

winning set W of the COSYNNC neural network controller. Both of these controllers are synthesized using the same quantization parameters, which results in similar abstractions. The completeness k is mathematically defined as:

$$k = \frac{|W|}{|W_s|} \tag{4-1}$$

The completeness represents to what extend the neural network controller encodes the same information as the traditional correct-by-design controller. It is also a measure of how close the neural network controller is to obtaining the largest obtainable winning set given the synthesis parameters.

For the neural network encoding the winning set W (W NN) similar completeness metrics are relevant. The extent to which the neural network encoding the winning set encompasses the entire winning set will be denoted by its completeness k_W defined as:

$$k_W = \frac{|X_w| + |X_l|}{|X_a|} \tag{4-2}$$

where $X_w = \{x \in X_a \mid x \in W \cap \vec{y}_2 \geq \sigma\}$ and $X_l = \{x \in X_a \mid x \in W^c \cap \vec{y}_1 \geq \sigma\}$ with $\vec{y} = f_{WNN}(\vec{x}, \boldsymbol{\theta}_{WNN})$ as defined in Definition 2-1-1 and σ the significance that the neural network needs to output to be taken as the truth. If this metric becomes 1 it implies that the neural network successfully encodes the subset of the state space X_a for which the states are in the winning set W and the subset for which they are outside of the winning set W^c. If k_W is not 1 it implies that the neural network is not encoding parts of the winning set or encoding them wrongly. In that case it is relevant to know how many false positives the neural network is providing. These are states for which the neural network encoding the winning set claims they are inside the winning set while they are actually not. If a neural network that encodes the winning set is used, for which its completeness k_W is not 1, it is important to verify that it at least does not provide any false positives. The number of false positives with respect to the cardinality of the state space k_{FP} is defined as:

$$k_{FP} = \frac{|X_f|}{|X_a|} \tag{4-3}$$

where $X_f = \{x \in X_a \mid x \in W^c \cap \vec{y}_2 \geq \sigma\}$ with $\vec{y} = f_{WNN}(\vec{x}, \boldsymbol{\theta}_{WNN})$ as defined in Definition 2-1-1 and σ the significance that the neural network needs to output to be taken as the truth. These two metrics and the data requirement for the neural network encoding the winning set (W NN) will allow for a thorough comparison between correct-by-design neural network controllers and traditional binary decision diagram stored correct-by-design controllers.

All of these metrics will allow for useful insight to be gained about the effectiveness of correct-by-design neural network controllers and how they compare to their traditional counterpart. Since the amount of data required to store correct-by-design controllers is a major limitation in correct-by-design control, the data related metrics could provide valuable insights into the deployability of neural network controllers.

4-2 Linear systems

In this section, linear systems will be considered and synthesized into correct-by-design neural network controllers. These controllers will then be compared to traditional correct-by-design controllers.

4-2-1 Single-input single-output systems

First, a SISO system will be considered that models a rocket that is fighting the force of gravity. The rocket can only travel on a one-dimensional axis and the position and velocity of the rocket make up the state of the rocket. The state space is therefore planar. The dynamics of the rocket are mathematically described as:

$$\dot{\vec{x}} = \begin{bmatrix} \dot{y} \\ \dot{v} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} y \\ v \end{bmatrix} + \begin{bmatrix} 0 \\ \frac{1}{M} \end{bmatrix} u + \begin{bmatrix} 0 \\ -g \end{bmatrix} \tag{4-4}$$

where y represents the position on its axis of freedom, v the velocity on that axis, M the mass of the rocket and g the acceleration the rocket experiences due to the force of gravity. The input u represents a force in N that is being exerted on the rocket due to the rocket engines firing. The firing of the rocket engines is the control input of the system.

A number of controllers can now be synthesized in order to make the feedback controlled system adhere to a set based predefined specification. A straight forward specification would be to have the controller fly the rocket to the equilibrium point where its position is $y = 0$ and $v = 0$. This will be modelled by specifying a reachability control specification with the to be reached set Z defined as all the points $x \in X$ where $y \in [-1, 1]\ m$ and $v \in [-1, 1]\ \frac{m}{s}$. The mass of the rocket is taken to be $M = 267$ kg and the acceleration due to gravity $g = 9.81\frac{m}{s^2}$

For the abstraction, the state space of the systems is restricted to $y \in [-5, 5]\ m$ and $v \in [-10, 10]\ \frac{m}{s}$ with the state quantization parameter $\vec{\eta} = [0.1\ m, 0.1\ \frac{m}{s}]^T$. These parameters result in a state space which contains 20301 states. The input space is defined as $u \in [0, 6000]\ N$ with the quantization parameter $\eta_u = 1000\ N$. The topology that is used for the neural network controller is a multilayer perceptron network with a $2 - 12 - 8 - 7$ topology where the hidden layer depth is thus 2. The network uses ReLU activation functions. For this example, the training focuses radial outwards, losing states and all states are used. The sampling time is set to $\tau = 0.1\ s$, the finite horizon to $N = 50$ and $\lambda = 0.0075$.

Synthesizing the controller results in a controller with a winning set W containing 76.7992% of the bounded state space. The total synthesis time for this controller is NNt = 2842 s. The winning set and a feedback controlled walk of the plant is depicted in Figure 4-1. Plotting the partitions of the neural network reveals how the neural network controller controls the system. The partitions are depicted in Figure 4-2.

Although these plots proof that neural network controllers can indeed be used to control such linear systems, they do not boast a lot of meaningful information unless they are compared to a valid baseline. For this purpose, the metrics as presented in Table 4-1 are evaluated.

Synthesizing a correct-by-design controller in SCOTS yields a completeness of $k = 100\%$ since both the SCOTS controller and the neural network controller boast a winning set size W of 76.7992%. The synthesis procedure uses the same linear transitions as calculated by COSYNNC. The total synthesis time for the SCOTS controller is BDDt = 179 s. The SCOTS controller is synthesized using the same abstraction parameters as the COSYNNC controller.

Furthermore, a neural network is trained to encode the winning set W of the correct-by-design neural network controller. This neural network is multilayer perceptron of a $2 - 8 - 8 - 2$

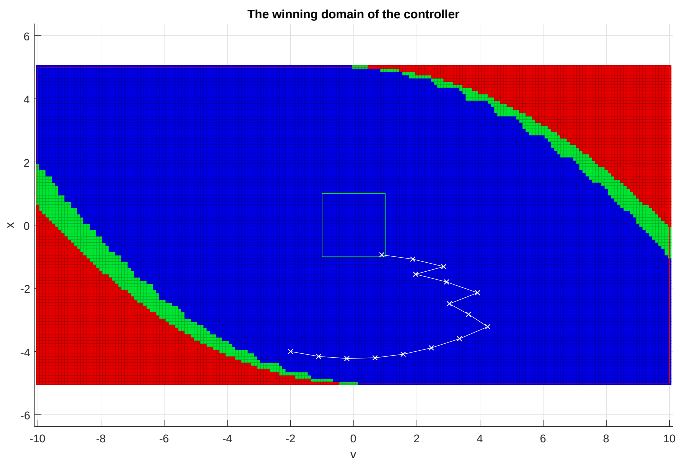

Figure 4-1: The winning set W of the correct-by-design neural network controller of the rocket system for a reachability specification. The blue hyper-cells are hyper-cells in the winning set W, the red hyper-cells and green hyper-cells are hyper-cells outside of the winning set. The green hyper-cells are hyper-cells which do not have a formal guarantee but for which the state in the center does reach the to be reached set given a finite horizon. The white trace represents a feedback controlled walk to demonstrate the performance of the controller.

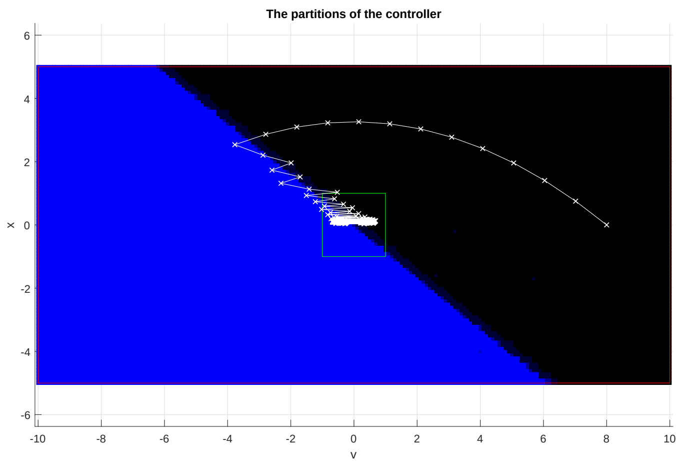

Figure 4-2: The partitions that the controller creates. The colored areas represent a particular input being fed to the plant given that the plant is in that subset of the state space. The white trace represents a feedback controlled walk to demonstrate the performance of the controller.

Table 4-3: The required amount of data per structure for the reachability rocket controller.

Structure	Data requirement	Unit
NN	781	#bytes
W BDD	1813	#bytes
NN + W BDD	2594	#bytes
W NN	461	#bytes
NN + W NN	1242	#bytes
NN BDD	3014	#bytes
BDD	13791	#bytes

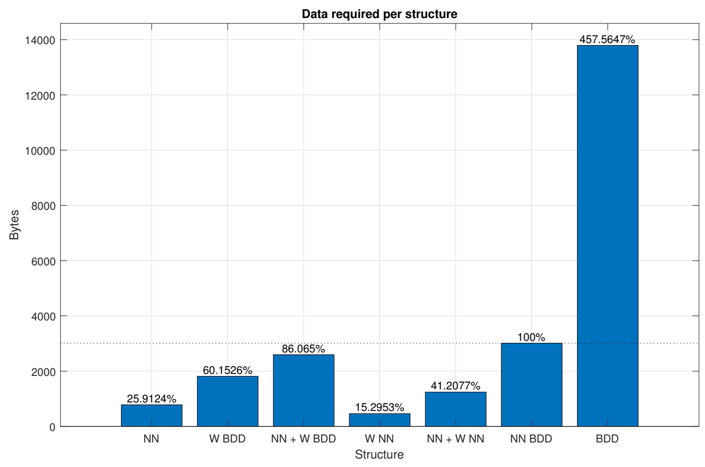

Figure 4-3: An overview of the different controllers and winning set representations in terms of their data requirement in bytes for the reachability rocket controller. The definition of the different structures is provided in Section 4-1.

topology and encodes $k_W = 97.01\%$ of the winning set W. It does so while providing $k_{FP} = 0.00\%$ false positives. The training time required to train this neural network encoding the winning set is WNNt = 6021 s.

A table of the required data in bytes is depicted in Table 4-3. A graphical representation of the data required per structure and how this relates to the NN BDD structure is depicted in Figure 4-3.

4-2-2 Multiple-input multiple-output systems

Another linear system will be considered that is a multiple-input multiple-output (MIMO) system. The system is simply a randomly generated 2x2 matrix with two different input dimensions in order to yield a MIMO system. The system can be described using a state-space representation as:

$$\dot{\vec{x}} = \begin{bmatrix} 3.8045 & 0.7585 \\ 5.6782 & 0.5395 \end{bmatrix} \vec{x} + \begin{bmatrix} 5.3080 & 9.3401 \\ 7.7917 & 1.2991 \end{bmatrix} \vec{u} \tag{4-5}$$

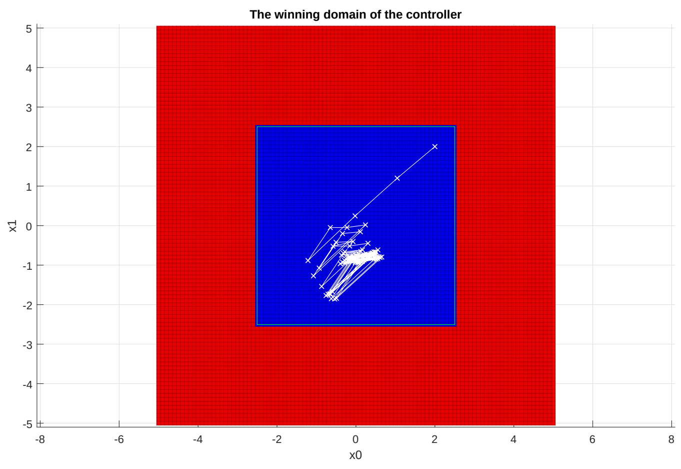

Figure 4-4: The winning set W of the correct-by-design neural network controller of the MIMO system for an invariance specification. The blue hyper-cells are hyper-cells in the winning set W, the red hyper-cells are hyper-cells outside of the winning set. The white trace represents a feedback controlled walk to demonstrate the performance of the controller.

The poles of the uncontrolled system $(u = [0, 0]^T)$ lie at $\lambda_1 = 4.8124$ and $\lambda_2 = -0.4684$. The system is therefore unstable if left uncontrolled.

A neural network controller will be synthesized for this MIMO system to adhere to an invariance control specification. The state space is first restricted to $\vec{x} \in [-5, 5] \times [-5, 5]$ with state quantization parameter $\vec{\eta} = [0.1, 0.1]^T$. These parameters result in a state space which contains 10201 states. The input space is restricted to $\vec{u} \in [-5, 5] \times [-2.5, 2.5]$ with input quantization parameter $\vec{\eta}_u = [5, 2.5]^T$. The quantization of the input space is intentionally made very rough to prevent the input space from becoming very large. The controller is tasked with making sure that the state of the plant stays in the invariant set $\vec{x} \in [-2.505, 2.505] \times [2.505, 2.505]$. The neural network controller is a multilayer perceptron with a $2-8-8-9$ topology and labelled output neurons. The network uses ReLU activation functions. The employed training focuses are radial outwards and neighboring losing states. The sampling time is set to $\tau = 0.025\ s$, the finite horizon to $N = 25$ and $\lambda = 0.0075$.

After NNt $= 2328\ s$ of running the synthesis routine the verification routine returns a controller that has a winning set of 25.4975%. This is equivalent to the entire invariant set, signifying that the neural network controller has found the appropriate control inputs to make sure the plant stays in the invariant set. A plot of the winning set is depicted in Figure 4-4. The partitions that are formed by the neural network controller is visualized in Figure 4-5.

In order to provide a comparison in terms of the required data for this MIMO neural network controller, a comparison is again made to a conventional correct-by-design controller synthesized using SCOTS. Synthesizing the SCOTS controller using the same abstraction parameters results in the same winning set that contains 25.4975% of the state space. The synthesis procedure uses the same the linear transitions as calculated by COSYNNC. The total synthesis time for the SCOTS controller is BDDt $= 99$ s. The completeness of this neural network controller is therefore $k = 100\%$.

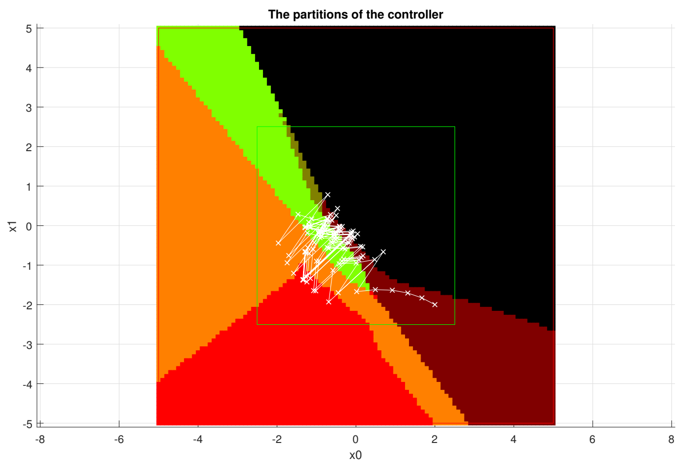

Figure 4-5: The partitions that the controller creates. The colored areas represent a particular input being fed to the plant given that the plant is in that subset of the state space. The white trace represents a feedback controlled walk to demonstrate the performance of the controller.

Table 4-4: The required amount of data per structure for the MIMO invariance controller.

Structure	Data requirement	Unit
NN	713	#bytes
W BDD	701	#bytes
NN + W BDD	1414	#bytes
W NN	461	#bytes
NN + W NN	1174	#bytes
NN BDD	1197	#bytes
BDD	3442	#bytes

A neural network is also trained to encode the winning set W of the correct-by-design neural network controller. This neural network is a multilayer perceptron of a $2-8-8-2$ topology and encodes $k_W = 98.49\%$ of the winning set W. It does so while providing $k_{FP} = 0.00\%$ false positives. The training time required to train this neural network encoding the winning set is WNNt $= 4819$ s.

The amount of data required per structure is presented in Table 4-4. A graphical overview of the data required per structure and how this relates to the NN BDD structure is provided in Figure 4-6.

4-3 Linear hybrid systems

In order to demonstrate that the methodology is also applicable to linear hybrid systems a DC-to-DC converter circuit will be considered. The DC-to-DC converter circuit has two modes. The mode in which the circuit is operating is based on whether or not a certain switch is opened or closed. Operation of this switch will be left to the controller. By appropriately

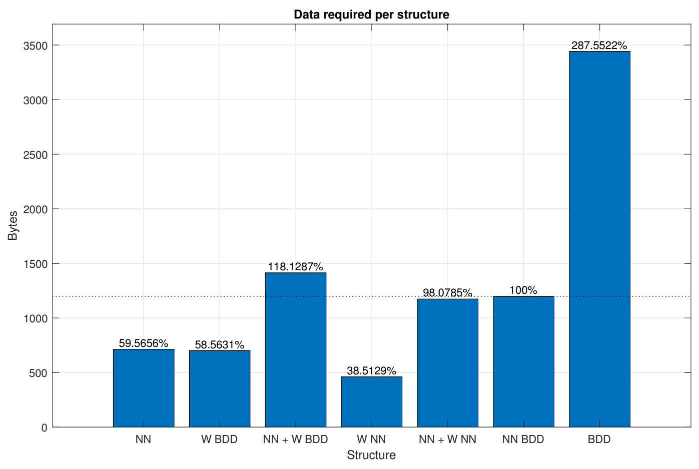

Figure 4-6: An overview of the different controllers and winning set representations in terms of their data requirement in bytes for the MIMO plant. The definition of the different structures is provided in Section 4-1.

opening and closing this switch the output voltage and current of the circuit can be regulated to the desired levels.

The DC-to-DC circuit exhibits linear dynamics in each of the modes but the modes are switched. Hence the model is a linear hybrid model. The model that describes the circuit can be mathematically described as:

$$\dot{\vec{x}} = \begin{bmatrix} \dot{i} \\ \dot{v} \end{bmatrix} = \begin{cases} \begin{bmatrix} \frac{-r_l}{x_l} & 0 \\ 0 & -\frac{1}{x_c}\frac{1}{r_0+r_c} \end{bmatrix} \vec{x} + \begin{bmatrix} \frac{v_s}{x_l} \\ 0 \end{bmatrix} & u = 1 \\ \\ \begin{bmatrix} -\frac{1}{x_l}(r_l + \frac{r_0 r_c}{r_0+r_c}) & -\frac{1}{5}\frac{1}{x_l}\frac{r_0}{r_0+r_c} \\ 5\frac{r_0}{r_0+r_c}\frac{1}{x_c} & -\frac{1}{x_c}\frac{1}{r_0+r_c} \end{bmatrix} \vec{x} + \begin{bmatrix} \frac{v_s}{x_l} \\ 0 \end{bmatrix} & \text{otherwise} \end{cases} \tag{4-6}$$

where r_l, r_0, x_l and x_c are parameters based on the values of the components in the circuit and u is the input that determines in which mode the circuit is operating.

Similarly to the rocket example, a controller is synthesized based on an abstraction of this system. The state space will be restricted to states where $i \in [0.65, 1.65]$A and $v \in [4.95, 5.95]$V with state quantization parameter $\vec{\eta} = [0.005\text{A}, 0.005\text{V}]^T$. These parameters result in a state space which contains 40401 states. The input space consists of $u = \{1, 2\}$. The control specification is a reachability specification with the to be reached set Z defined as $i \in [1.1, 1.6]$A and $v \in [5.4, 5.9]$V. The neural network controller is a multilayer perceptron neural network with a $2-8-8-2$ topology and labelled output neurons. The network uses ReLU activation functions. The employed training focuses during synthesis are radial outwards and all states. The sampling time of the plant is set to $\tau = 0.5$ s, the finite horizon to $N = 50$ and $\lambda = 0.0075$.

Synthesizing a neural network controller using these parameters resulted in a controller that can control 92.83% of the state space. This result was obtained after running the procedure

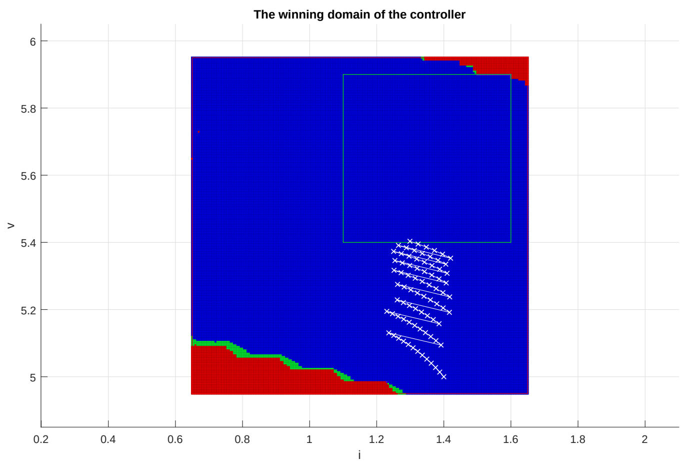

Figure 4-7: The winning set W of the correct-by-design neural network controller of the DC-to-DC circuit system for a reachability specification. The blue hyper-cells are hyper-cells in the winning set W, the red hyper-cells and green hyper-cells are hyper-cells outside of the winning set. The green hyper-cells are hyper-cells which do not have a formal guarantee but for which the state in the center does reach the to be reached set given a finite horizon. The white trace represents a feedback controlled walk to demonstrate the performance of the controller.

for NNt $= 1394\ s$. The winning set of the feedback controlled plant is depicted in Figure 4-7. The partitions that this particular controller creates are depicted in Figure 4-8.

Synthesizing a conventional correct-by-design controller using SCOTS with the same synthesis parameters results in a controller with a winning set of 93.6% of the state space. The resulting neural network controller therefore has a completeness of $k = 99.18\%$. The synthesis procedure uses the same the linear transitions as calculated by COSYNNC and takes a total of BDDt $= 380\ s$. The resulting binary decision diagram stored controller was also determinized to form a determinized binary decision diagram controller.

In addition to this, a neural network is trained to encode the winning set W of the correct-by-design neural network controller. This neural network is a multilayer perceptron of a $2-8-8-2$ topology and encodes $k_W = 97.97\%$ of the winning set W. It does so while providing $k_{FP} = 0.00\%$ false positives. The training time required to train this neural network encoding the winning set is WNNt $= 7176\ s$.

An overview of the amount of data required per structure is depicted in Table 4-5. An overview of the data required by the different structures and how they compare to the NN BDD structure is represented in Figure 4-9.

4-4 Nonlinear systems

To demonstrate that the methodology is also applicable to nonlinear systems, a nonlinear system is considered. The nonlinear system will represent a unicycle traveling at a constant speed v through a planar landscape. The input to the system consists of the steering actions $u = \{\mathit{left}, \mathit{straight}, \mathit{right}\}$ represented by a signed integer $u = \{-1, 0, 1\}$. The goal of the

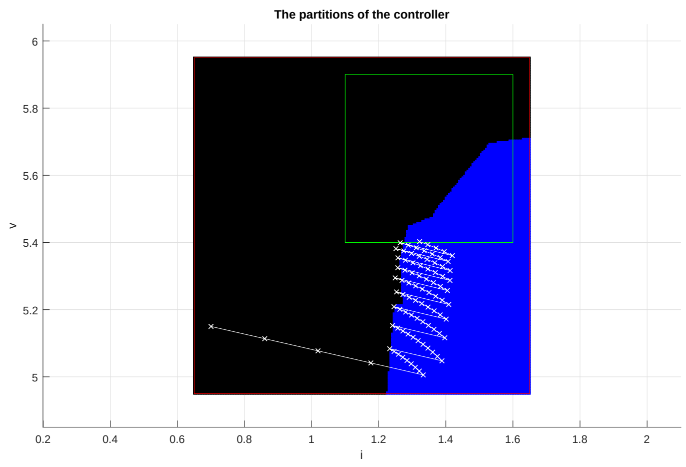

Figure 4-8: The partitions that the controller creates. The colored areas represent a particular input being fed to the plant given that the plant is in that subset of the state space. The white trace represents a feedback controlled walk to demonstrate the performance of the controller.

Table 4-5: The required amount of data per structure for a DC-to-DC reachability controller.

Structure	Data requirement	Unit
NN	461	#bytes
W BDD	1030	#bytes
NN + W BDD	1491	#bytes
W NN	461	#bytes
NN + W NN	922	#bytes
NN BDD	3076	#bytes
D BDD	1470	#bytes
BDD	4095	#bytes

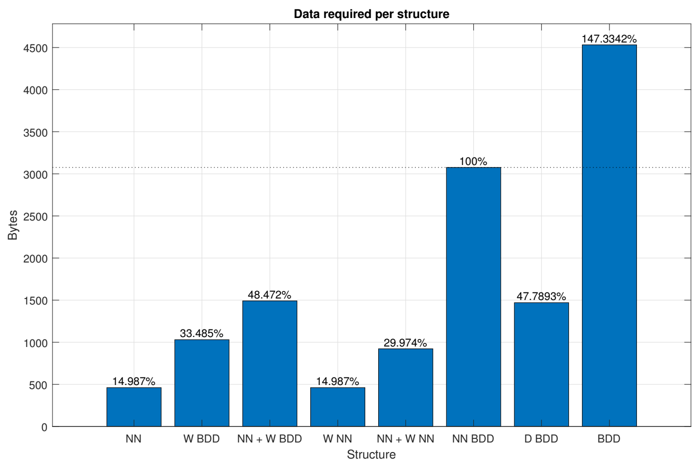

Figure 4-9: An overview of the different controllers and winning set representations in terms of their data requirement in bytes for the DC-to-DC circuit system. The definition of the different structures is provided in Section 4-1.

controller is to provide the appropriate steering actions to the unicycle, independent of the location, such that the unicycle is steered to a target set Z in the state space.

The dynamics of the nonlinear unicycle are mathematically described by:

$$\dot{\vec{x}} = \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} v\cos(\theta) \\ v\sin(\theta) \\ u\omega \end{bmatrix} \tag{4-7}$$

where v represents a constant speed at which the unicycle is traveling and ω represents the rate at which the angle of the unicycle changes due to a steering input. Due to the sin and cos term, the dynamics of the unicycle are nonlinear. This nonlinearity requires that the system is provided with a radial growth bound function which is used to over-approximate the dynamics of the plant. The radial growth bound function used is mathematically described by:

$$\dot{\vec{r}} = \begin{bmatrix} \dot{r}_x \\ \dot{r}_y \\ \dot{r}_\theta \end{bmatrix} = \begin{bmatrix} r_\theta v \\ r_\theta v \\ 0 \end{bmatrix} \tag{4-8}$$

Using this radial growth bound, it is now possible to synthesize a correct-by-design neural network controller.

To synthesize a controller, the appropriate synthesis parameters must be defined. The state space of the unicycle will be restricted to $x \in [-5,5] \times [-5,5] \times [0,2\pi]$ with state quantization parameter $\vec{\eta} = [0.1, 0.1, 0.1]^T$. These parameters result in a state space which contains 642663 states. The input space will be restricted to the set $u = \{-1, 0, 1\}$. The neural network controller is a multilayer perceptron with a $3-12-12-3$ topology and labelled output neurons. The network uses ReLU activation functions. The training focuses used during synthesis are radial outwards, all states and losing states. The sampling time is set to $\tau = 0.3\ s$, the finite horizon to $N = 50$ and $\lambda = 0.0075$.

Synthesizing a neural network controller using these parameters resulted in a controller that can control 79.5% of the state space. This result was obtained after running the synthesis procedure for NNt $= 5612\ s$. The winning set of the feedback controlled system is depicted in Figure 4-10 for a slice of the state space at $\theta = 3.0$. The partitions as created by the controller are depicted in Figure 4-11.

For comparison, a SCOTS controller is synthesized using the same abstraction parameters. The synthesis time for the SCOTS controller amounts to BDDt $= 136\ s$. The resulting SCOTS controller has a winning set that covers 93.73% percent of the finite cardinality state space. The completeness of the neural network controller is therefore $k = 84.87\%$. The binary decision diagram stored controller was also determinized to form a determinized binary decision diagram controller.

Furthermore, a neural network is trained to encode the winning set W of the correct-by-design neural network controller. This neural network is a multilayer perceptron of a $3 - 24 - 36 - 36 - 24 - 2$ topology and encodes $k_W = 70.05\%$ of the winning set W. It does so while providing $k_{FP} = 1.89\%$ false positives. Due to the large cardinality of the state space the training was performed on a different system. The neural network was trained on a Intel Xeon W-2145 3.7 GHz CPU with 32 GB of RAM. The training time required to train this neural network encoding the winning set is WNNt $= 5523\ s$. It should be noted that the

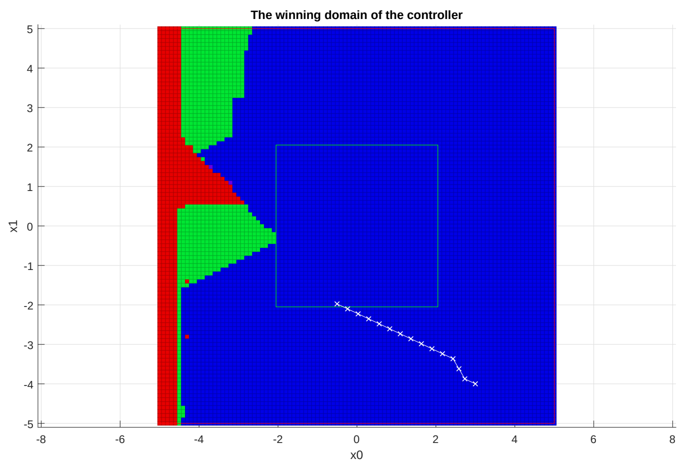

Figure 4-10: The winning set W of the correct-by-design neural network controller of the unicycle system for a reachability specification. The figure represents a slice in the state space where $\theta = 3.0$. The blue hyper-cells are hyper-cells in the winning set W, the red hyper-cells and green hyper-cells are hyper-cells outside of the winning set. The green hyper-cells are hyper-cells which do not have a formal guarantee but for which the state in the center does reach the to be reached set given a finite horizon. The white trace represents a feedback controlled walk to demonstrate the performance of the controller.

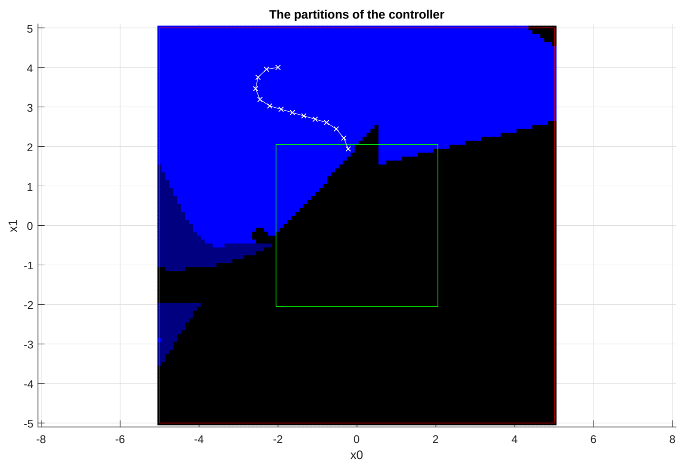

Figure 4-11: The partitions that the controller creates for the unicycle system. The figure represents a slice in the state space where $\theta = 3.5$. The colored areas represent a particular input being fed to the plant given that the plant is in that subset of the state space. The white trace represents a feedback controlled walk to demonstrate the performance of the controller.

Table 4-6: The required amount of data per structure for the unicycle reachability controller.

Structure	Data requirement	Unit
NN	977	#bytes
W BDD	52432	#bytes
NN + W BDD	53409	#bytes
W NN	13071	#bytes
NN + W NN	14048	#bytes
NN BDD	91447	#bytes
D BDD	16964	#bytes
BDD	43664	#bytes

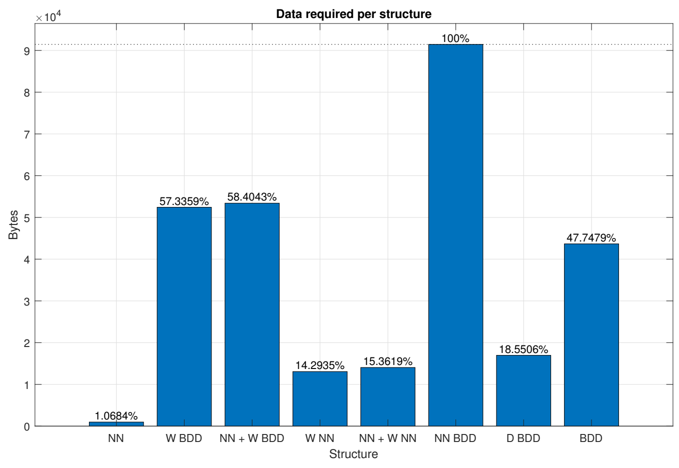

Figure 4-12: An overview of the different controllers and winning set representations in terms of their data requirement in bytes for the unicycle system. The definition of the different structures is provided in Section 4-1.

controller only encodes a portion of the entire winning set and is therefore not very useful in practice.

An overview of the amount of data required per structure is depicted in Table 4-6. The same information is depicted in a graphical overview in Figure 4-12.

Table 4-7: An overview of the data requirements for all the plants and their relevant structures. #B abbreviates number of bytes. Σ represents the sum of the neural network controller (NN) and the neural network encoding the winning set (W NN). The meaning of all the structure abbreviations is given in Table 4-1.

Plant	NN [#B]	k [-]	W NN [#B]	$k_W[-]$	Σ [#B]	NN BDD [#B]	D BDD [#B]	BDD [#B]
Rocket	781	100%	461	97.01%	1242	3014	-	13791
MIMO	713	100%	461	98.49%	1174	1197	-	3442
DC-to-DC	461	99.18%	461	97.97%	922	3076	1470	4532
Unicycle	977	84.87%	13071	70.05%	14048	91447	16964	43664

Table 4-8: An overview of the synthesis time for all of the controllers. The meaning of the abbreviations is given in Table 4-2.

Plant	NNt [s]	WNNt [s]	BDDt [s]
Rocket	2842	6021	179
MIMO	2328	4819	99
DC-to-DC	1394	7176	380
Unicycle	5612	5523	136

4-5 Summary

In this section, the most important results from the previous sections are presented in a summary consisting of a few tables in order to present the experimental results in a more succinct manner.

An overview of the data requirements and completeness k for the correct-by-design neural network controllers, the traditional correct-by-design controllers and for the winning sets encoded as a neural networks for all the different plants is given in Table 4-7. The meaning of the structure abbreviations is provided in Table 4-1. The meaning of the neural network related completeness properties is provided in Section 4-1.

An overview of the synthesis time for the correct-by-design neural network controller, the neural network encoding the winning set and the traditional correct-by-design controller for all the different plants is given in Table 4-8[1]. For clarity it should again be iterated that the other structures that are presented in this thesis work are conversions or simplifications of these two fundamental structures with trivial conversion times. Therefore, the synthesis times for these two fundamental structures are the only relevant metrics.

[1]It should be noted that the time for the winning set neural network (W NN) for the unicycle was obtained using a different device for training and is hence not representative

Chapter 5

Discussion

In this chapter the methodology as proposed in Chapter 3 and the results as presented in Chapter 4 are discussed and evaluated.

In this thesis work, the goal was to explore the possibilities with regards to the synthesis of correct-by-design controllers. Based on the developed methodology and the results, it can be concluded that correct-by-design neural network controllers can be synthesized, verified and perform appropriately as is demonstrated by the variety of controllers presented in Figure 4-1, 4-7, 4-4, 4-10. These same results also demonstrate that it is possible to synthesize such a controller for both linear and nonlinear dynamics. In addition to this, the proposed methodology uncovered an extension for feedback refinement relation based abstractions for linear systems based on their linear dynamics which is less conservative than its nonlinear counterpart.

From Figure 4-3, 4-4, 4-5, 4-6 and Table 4-7 one can conclude that, in general, the resulting neural network controllers are significantly smaller in terms of their data requirement than their traditional correct-by-design controller counterparts. A few observations can be made that support and explain this claim. First of all, it should be noted that the neural network controller does not store inputs for every state by storing all the states in the winning set but rather, maps states to inputs based on its weights and biases. By not storing all the states in the winning set, a significant amount of data is freed up. Neural network controllers therefore do not suffer the curse of dimensionality with respect to the state space. This also implies that refining the state space by decreasing the state quantization parameter η does not result in larger neural network controllers. Neural network controllers are therefore robust with respect to larger state spaces. Secondly, it should be noted that the neural network controllers are, by their very nature, deterministic during operating whereas traditional correct-by-design controller do not need to be deterministic. Since nondeterminism requires storing a set of inputs for every state in the winning set rather than just a single input, it requires more data to store them. For these reasons, neural network controllers are generally significantly smaller than traditional correct-by-design controllers in terms of their required data.

With respect to the input space cardinality, it should be noted that neural network controllers are not necessarily robust. For the labelled neural network controller topology, since every input in the input space needs to be represented by an individual output neuron, the size of the neural network controller quickly grows with larger input spaces. The advantage of using this topology however is that in practice, it trains well and is often able to find the appropriate inputs due to the nature of the policy gradients that are inherent to this topology. The unlabelled neural network controller topology does not suffer the curse of dimensionality for the input space, since each extra dimension in the input space only adds two extra output neurons. Furthermore, the number of neurons also does not increase for finer quantization parameters η_u. However, contrary to the labelled topology, the policy gradients are a lot less pronounced and training therefore takes a lot longer. In addition to this, the unlabelled topology also imposes hard restrictions on the search space within the input space during training. This is a direct result of how the unlabelled neural network controller topology is constructed and limits its effectiveness.

If a representation of the winning set is a hard requirement, the discussion with respect to the amount of required data becomes more nuanced. The results show that it is indeed possible to synthesize correct-by-design neural network controllers which, accompanied by a binary decision diagram representation of the winning set W (NN + W BDD), are still smaller in terms of their data requirements than a conventional correct-by-design controller (BDD). Figure 4-3 and 4-9 show that they can also be smaller than a binary decision diagram representation of the neural network controller (NN BDD). This is however not guaranteed as can be seen in Figure 4-6 and 4-12. This can largely be accredited to the way in which binary decision diagrams actually compress. Their compressive capability stems from the structure of the binary atoms that make up the binary function. If this structure is favourable the entire binary function will contain large redundancies and is therefore able to compress, if this is not the case it will have bad compressive capabilities. Binary decision diagrams that encode more nondeterminism have a larger probability of containing these redundancies and will therefore sometimes boast large compressive capabilities due to the nature of the data structure.

Instead of using a binary decision diagram to encode the winning set, neural networks have also been used to encode the winning set. This type of problem is an optimization problem where the neural network is tasked with encoding the largest possible portion of the winning set without providing false positives. As can be seen in the Table 4-7, these winning sets become slightly smaller than the actual winning set in order to ensure that they do not contain any false positives. The table also shows that encoding the winning set as a neural network indeed results in a smaller representation than a binary decision diagram that encodes the winning set in terms of the data required to do so. Using this type of representation can therefore significantly compress the data needed to encode the winning set of the controller at the cost of a slightly smaller winning set. It should however be noted that it may not always be possible or feasible to construct such a neural network encoded winning set. An example of this is the unicycle example as presented in Section 4-4. The reason why this winning set does not encode as well as the other winning sets is likely because of the large state space cardinality and the nature of the corresponding winning set. Since the winning set is somewhat sporadic and the encoder weighs every state equally it is not able to encode the majority of the state space without introducing false positives. Since false positives are to be avoided the resulting winning set is only a portion of the actual winning set. In practice this could be resolved by introducing more neurons so that the neural network may store more information. This could also allow it to better store the sporadic nature of the winning set.

In practice this may however make the neural network encoded winning sets less competitive than binary decision diagram stored winning sets.

A comparison between the combined neural network controller and binary decision diagram of the winning set (NN + W BDD) and a determinized SCOTS controller stored as a binary decision diagram (D BDD) is also made in terms of the required data. In Table 4-5, it can be seen that the two metrics differ by only 21 bytes in favour of the determinized SCOTS controller and are hence very close in terms of data. When considering the unicycle system, as depicted in Table 4-6, it can be seen that, when compared to the combined data requirement, the determinized SCOTS controller outperforms the neural network controller. It should however be noted that whenever the winning set is not a hard requirement, the correct-by-design neural network controllers still easily outperform the determinized SCOTS controllers in terms of data.

With regards to the synthesis time for correct-by-design neural network controllers with respect to traditional correct-by-design controllers such as the ones generated with SCOTS, it is evident that the traditional correct-by-design controllers decisively outperform the neural network controllers. This is clearly illustrated by the overview of the synthesis times for different controllers given in Table 4-8. This is largely due to the fact that the operations required to synthesize a traditional correct-by-design controller are also executed during the correct-by-design neural network controller synthesis procedure. In addition to those operations, the neural network also needs to be trained and the plant needs to be simulated. This results in significantly longer synthesis times. Furthermore, the neural network controller synthesis procedure boasts less guarantees with regards to terminating with a nontrivial controller. The correct-by-design neural network synthesis control procedure also requires more user input with respect to the parameters that are to be used during synthesis. The effect of the parameters is also quite large which makes synthesizing such controllers a somewhat iterative process, further increasing the time that is dedicated to the process.

One final observation that can be made with regards to the comparison between correct-by-design neural network controllers and traditional correct-by-design controllers is that the neural network controllers are computationally cheap to operate on-line. This is due to the fact that they only require elementary mathematical operations in order to be evaluated. This is in contrast to the binary decision diagram stored controllers that require external binary decision diagrams libraries, such as the CUDD library, to read out. The addition of such libraries in order to evaluate the binary decision diagrams can significantly add to the amount of data and computational power required to operate these types of controllers. It could therefore be that neural networks are simply quicker and computationally cheaper to evaluate when compared to binary decision diagrams for certain hardware architectures. Because of this, neural network controllers are indeed a valid option in certain environments such as ones that require a high sampling time, have low computational power and/or have a limited amount of memory available to them.

Based on these findings, one can determine some general rules of thumb regarding the use of the different control structures considered in this thesis work. If the initial state set of the plant is known a priori or the winning set is all together not required correct-by-design neural network controllers significantly outperform traditional binary decision diagram stored correct-by-design controllers in terms of their data requirements. The same is also true if very fine state quantization is a requirement as the amount of data required by the neural network controllers does not scale with the state quantization parameters used for the abstractions. Finally, if the amount of memory on the device implementing the controller is severely limited, correct-by-design neural network controllers also outperform traditional correct-by-design controllers. This is due to the fact that neural network controllers are, in general, significantly smaller in terms of their required data than traditional correct-by-design controllers. Furthermore, the neural network controllers only require elementary mathematically operations to be read out. This is in contrast to binary decision diagrams that require additional libraries and thus memory in order to be read out and manipulated. If memory is not an issue, it is preferable to use the traditional correct-by-design controllers. The reason for this is the fact that they are significantly faster to synthesize and will, in general, provide a fuller solution. This can be accredited to the exhaustive nature of the synthesis method used to synthesize these types of controllers. Traditional correct-by-design controllers are also preferred in case the input space is large, as the size of the neural network controllers quickly grow with larger input spaces when using the labelled topology. Although the unlabelled neural network controller topology could theoretically alleviate this problem, it has not been found to train well in practice and therefore does not outperform traditional correct-by-design controllers in this front.

Chapter 6

Conclusion

In this chapter the conclusions that can be drawn from this thesis work are presented. In addition to this, future work that can be done in the context of this thesis will be presented.

6-1 Conclusion

In this thesis work a novel methodology to synthesize correct-by-design neural network controllers has been developed. The developed methodology combines the use of traditional correct-by-design control techniques and neural network techniques. The methodology consists of having a neural network controller interact in a closed loop simulated environment with the plant and appropriately reinforcing the resulting control policy. This reinforcing of the neural network's control policy is done based on the performance of finite horizon episodes that simulate its behaviour. After an arbitrary amount of training, the neural network controller is verified using feedback refinement relation based abstractions and fixed-point algorithms. This routine yields the winning set for which the controller acts appropriately which in turn is used to refine the training routine. This procedure will result in a correct-by-design neural network controller with guarantees on its winning set.

In this work, the proposed methodology has been used to synthesize correct-by-design neural network controllers. This has shown that the methodology indeed works and is applicable to a variety of different system classes such as linear systems, linear hybrid systems, nonlinear systems and nonlinear hybrid systems.

The resulting correct-by-design neural network controllers have been verified to work on their winning set and have been compared to conventional correct-by-design controllers in terms of their required amount of data. This comparison has showed that correct-by-design neural network controllers outperform conventional correct-by-design controllers in terms of their data requirements as long as a representation of the winning set is not required. If such a representation is required, correct-by-design neural network controllers can potentially still outperform the traditional controllers by encoding the winning set as a binary decision diagram or as a neural network. These neural network encoded winning sets will however generally encode a slightly smaller winning set due to the nature of the optimization problem these networks are trying to solve.

As part of this thesis, the proposed methodology for synthesis of correct-by-design neural network controllers was developed into a framework called COSYNNC. Using this framework, users can synthesize their own correct-by-design neural network controllers. In addition to this, the framework also has a variety of interfacing options for SCOTS so that users can make their own comparisons. This framework is publicly available and free to use and build upon.

6-2 Future work

With regards to the future work that can be done in the context of this research, there are a number of aspects that are worth pursuing.

First of all, other neural network topologies could be used and evaluated to determine if other neural network topologies are better suited to the task of performing correct-by-design control. Among the potential topologies are: convolutional neural networks, recurrent neural networks and long short-term memory neural networks.

Secondly, a better performance accreditation algorithm could be developed and implemented. This would allow for finer performance accreditation to specific inputs in the input chain of an episode. For example, a performance accreditation algorithm could be developed that estimates which state-input pair positively contributed to the outcome and which ones contributed negatively. By more finely accrediting the performance, the training routine could be made more effective resulting in better correct-by-design neural network controllers.

Another worthwhile addition would be the implementation of disturbed plants into the framework. This would allow for an even broader class of systems to be covered by the methodology.

Furthermore, alternative neural network controller topologies could be developed that are better suited towards higher dimensionality input spaces. Doing so would also further warrant the use of neural networks as correct-by-design controllers, since this would completely remove the curse of dimensionality from this particular domain.

Finally, the current version of the framework COSYNNC is only a proof of concept CPU based implementation. Since the methodology lends itself well to parallelization on CPU, GPU or even FPGA hardware, the entire synthesis procedure could be significantly sped up through parallelization.

Appendix A

COSYNNC Example

The required code to synthesize a correct-by-design reachability neural network controller consists of two parts. First the plant definition itself and secondly the code required to run the procedure.

A-1 Rocket plant definition

`Rocket.h`

```
#pragma once
#include "Plant.h"

namespace COSYNNC {
  class Rocket : public Plant {
  public:
    Rocket() : Plant(2, 1, 0.1, "Rocket", true) { }

    Vector DynamicsODE(Vector x, float t) override;
  private:
    const float _mass = 267; // kg
    const float _g = -9.81; // m s^-2
  };
}
```

`Rocket.cpp`

```
#include "Rocket.h"

namespace COSYNNC {
  Vector Rocket::DynamicsODE(Vector x, float t) {
    Vector dxdt = Vector(_stateSpaceDimension);

    dxdt[0] = x[1];
    dxdt[1] = _u[0]/_mass + _g;

    return dxdt;
  }
}
```

A-2 Procedure

```
Procedure cosynnc;

// Link the plant to the procedure
Plant* rocket = new Rocket();
cosynnc.SetPlant(rocket);

// Specify the state and input quantizers
cosynnc.SpecifyStateQuantizer(Vector({ 0.1, 0.1 }), Vector({ -5, -10 }),
    Vector({ 5, 10 }));
cosynnc.SpecifyInputQuantizer(Vector((float)5000.0), Vector((float)0.0),
    Vector((float)5000.0));

// Specify the synthesis parameters
cosynnc.SpecifySynthesisParameters(1000000, 50, 5000, 50000, 50);

// Link a neural network to the procedure
MultilayerPerceptron* multilayerPerceptron = new MultilayerPerceptron({
    8, 8 }, ActivationActType::kRelu, OutputType::Labelled);
multilayerPerceptron->InitializeOptimizer("sgd", 0.0075, 0.0);
cosynnc.SetNeuralNetwork(multilayerPerceptron);

// Specify the control specification
cosynnc.SpecifyControlSpecification(ControlSpecificationType::
    Reachability, Vector({ -1.0, -1.0 }), Vector({ 1.0, 1.0 }));

// Specify training focuses
cosynnc.SpecifyRadialInitialState(0.15, 0.85);
cosynnc.SpecifyTrainingFocus(TrainingFocus::RadialOutwards);

cosynnc.SpecifyTrainingFocus(TrainingFocus::LosingStates);

cosynnc.SpecifyTrainingFocus(TrainingFocus::AllStates);

// Specify synthesis parameters
cosynnc.SpecifyWinningSetReinforcement(true);

cosynnc.SpecifyUseRefinedTransitions(true);

// Specify where the controllers are saved
cosynnc.SpecifySavingPath("../controllers");

// Initialize the synthesize procedure
```

```
cosynnc.Initialize();

// Run the synthesize procedure
cosynnc.Synthesize();

// Free up memory
delete rocket;
delete multilayerPerceptron;
```

Bibliography

[1] Sheldon B. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, 27(6):509–516, 1978.

[2] Dirk Beyer. Graphical representation of a binary decision diagram.

[3] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[4] Dan Claudiu Cireşan, Ueli Meier, Luca Maria Gambardella, and Jürgen Schmidhuber. *Deep Big Multilayer Perceptrons for Digit Recognition*, pages 581–598. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[5] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12, 2011.

[6] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[7] Alex Graves, Abdel rahman Mohamed, and Geoffrey Hinton. Speech recognition with deep recurrent neural networks. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 6645–6649, May 2013.

[8] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9:1735–1780, 1997.

[9] Angga Irawan. Representing symbolic controllers with deep neural networks, 2018.

[10] Radoslav Ivanov, James Weimer, Rajeev Alur, George J. Pappas, and Insup Lee. Verisig: verifying safety properties of hybrid systems with neural network controllers. *Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control*, pages 169–178, 2019.

[11] Katarzyna Janocha and Wojciech Marian Czarnecki. On loss functions for deep neural networks in classification. *CoRR*, 2017.

[12] Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.

[13] Nal Kalchbrenner and Phil Blunsom. Recurrent continuous translation model. *Proceedings of the 2013 Conference of Empirical Methods in Natural Language Processing*, pages 1700–1709, Oct 2013.

[14] Bekir Karlik and A Vehbi Olgac. Performance analysis of various activation functions in generalized MLP architectures of neural networks. *Journal of Artificial Intelligence and Expert Systems*, 1, 2011.

[15] Andrej Karpathy. Deep reinforcement learning: Pong from pixels, May 2016.

[16] Diederik P. Kingma and Jimmy Lei Ba. Adam: A method for stochastic optimization. *3rd International Conference for Learning Representations*, Dec 2014.

[17] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[18] Steve Lawrence, C. Lee Giles, Ah Chung Tsoi, and Andrew D. Back. Face recognition: A convolutional neural network approach. *IEEE Transaction on neural networks*, 8, Jan 1997.

[19] Y. LeCun, B. Boser, J.S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1989.

[20] Yann LeCun, Corinna Cortes, and Christopher J.C. Burges. The MNIST database of handwritten digits.

[21] C Y Lee. Representation of switching circuits by binary decision programs. *Bell Systems Technical Journal*, 38, 1959.

[22] Yinan Li and Jun Liu. Rocs: A robustly complete control synthesis tool for nonlinear dynamical systems. In *Proceedings of the 21st International Conference on Hybrid Systems: Computation and Control (Part of CPS Week)*, HSCC '18, page 130–135, New York, NY, USA, 2018. Association for Computing Machinery.

[23] Manuel Mazo, Anna Davitian, and Paulo Tabuada. *PESSOA: A Tool for Embedded Controller Synthesis*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[24] Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, Dec 1943.

[25] Tomas Mikolov, Martin Karafiat, Lukas Burget, Jan "Honza" Cernock, and Sanjeev Khudanpur. Recurrent neural network based language model. *Interspeech 2010*, pages 1045–1048, 2010.

[26] Sebti Mouelhi, Antoine Girard, and Gregor Gössler. Cosyma: A tool for controller synthesis using multi-scale abstractions. In *Proceedings of the 16th International Conference on Hybrid Systems: Computation and Control*, HSCC '13, page 83–88, New York, NY, USA, 2013. Association for Computing Machinery.

[27] V Nair and G E Hinton. Rectified linear units improve restricted boltzmann machines. *Proceedings of the 27th International Conference on Machine Learning*, 27:807–814, Jun 2010.

[28] G. Reissig, A. Weber, and M. Rungger. Feedback refinement relations for the synthesis of symbolic controllers. *IEEE Transactions on Automatic Control*, 62(4):1781–1796, April 2017.

[29] Frank Rosenblatt. *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms*. Mar 1962.

[30] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323:533–536, 1986.

[31] Matthias Rungger and Majid Zamani. SCOTS: A tool for synthesis of symbolic controllers. *Proceedings of the 19th ACM International Conference on Hybrid Systems: Computation and Control*, 2016.

[32] P Smolensky. Information processing in dynamical systems: a foundation of harmony theory. *Parallel distributed processing: explorations in the microstructure of cognition*, 1:194–281, 1986.

[33] Paulo Tabuada. *Verification and Control of Hybrid Systems: A Symbolic Approach*. Springer, 2009.

[34] Ward van der Velden, Rob Neelen, Myla van Wegen, and Hero Miltenburg. Assessment of neural networks as correct-by-design controllers, 2017.

[35] Yuan Wang, Kirubakaran Velswamy, and Biao Huang. A long-short term memory recurrent neural network based reinforcement learning controller for office heating ventilation and air conditioning systems. *Processes*, 5, August 2017.

[36] Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, pages 229–256, 1992.

[37] Ivan S. Zapreev, Cees Verdier, and Manuel Mazo Jr. Optimal symbolic controllers determinization for bdd storage. *IFAC Conference on Analysis and Design of Hybrid Systems*, 51(16), 2018.

Glossary

List of Symbols

$\boldsymbol{\theta}$	Collection of the weights and biases of a neural network
g	Nonlinear activation function for a neural network neuron
λ	Gradient descent step size for the neural network backpropagation algorithm
$\boldsymbol{W}$	Weight matrix defining the connections between two layers
$\vec{b}$	Bias vector for a neural network layer
q	State quantization function
$\vec{\eta}$	Quantization parameter vector of the state space
q_u	Input quantization function
$\vec{\eta}_u$	Quantization parameter vector of the input space
$\vec{x}$	State of the plant
$\vec{\hat{x}}$	Quantized state of the plant
$\vec{x}_{nn}$	Normalized quantized state of the plant
$\vec{u}$	Input to the plant
$\vec{\hat{u}}$	Quantized input of the plant
$\vec{y}$	Output of the neural network
$\vec{d}$	Desired neural network output
$\mathcal{L}$	Loss function
S	System
X	State space set
X_0	Initial state space set
U	Input space set
$\rightarrow$	Transition function
Y	Output space set
H	Output function
Post_u	Post operator for input u
$\mathcal{B}_x$	Finite internal behaviour
$\mathcal{B}$	Finite external behaviour
$\mathcal{B}_x^\omega$	Infinite internal behaviour
$\mathcal{B}^\omega$	Infinite external behaviour
d	Metric function

n	Normalization function
n^{-1}	Denormalization function
γ	Probabilistic input sampling function
γ_g	Greedy input sampling function
X_a	Abstraction framework state space
U_a	Abstraction framework input space
Y_a	Abstraction framework output space
I_0	Training initial set
τ	Sampling time
X_p	Partial abstraction state space
U_p	Partial abstraction input space
ϕ	Plant dynamics
β	Growth bound function
X'	Transitionable state set
r	Radial growth bound
N	Episode finite horizon
W	Winning set
$\mathbb{B}$	A binary variable
$\wedge$	Binary AND operator
$\vee$	Binary OR operator
A	Ampere
V	Volt