COSYNNC: A Correct-by-Design Neural Network Controller Synthesis Framework

W. van der Velden

July 6, 2020

Contents

1 Introduction

This document is intended as a general user guide to the correct-by-design neural network controller synthesis framework COSYNNC. In this manual, a general overview of the framework will be given. This overview will then be further elaborated upon in order to provide more technical details. Furthermore, a small guide will be given with respect to the general usage of the framework along with some example code. Finally, an overview of the main functions of the synthesis framework will be provided.

2 Framework function

COSYNNC functions by combining reinforcement learning techniques with abstraction based verification techniques. For the abstraction based verification an abstraction is required that is of finite cardinality. For such an abstraction a formal system definition is required. A system S is defined, by Tabuada's definition [1], as:

$$S = (X, X_0, U, \rightarrow, Y, H) \tag{1}$$

- X is a non-empty set of states.
- $X_0 \subseteq X$ is a non-empty set of initial states.
- U is a non-empty set of inputs.

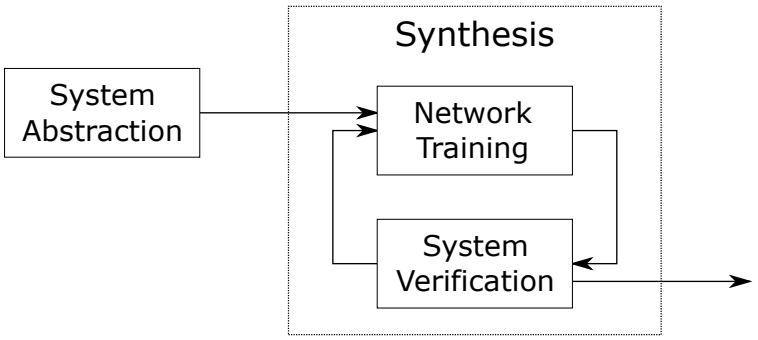

Figure 1: The different routines that make up COSYNNC.

- $\rightarrow \subseteq X \times U \times X$ is a transition relation between states and inputs to new states.
- Y is a non-empty set of outputs.
- $H : X \rightarrow Y$ is an output map.

In order to simplify the notation, the post operator is introduced such that $\text{Post}_u(x) = \{x' \in X \mid (x, u, x') \in \rightarrow\}$. 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 $S = (X, U, \rightarrow)$.

The framework consists of three different routines which are appropriately iterated in order to synthesis a correct-by-design neural network controller. These routines are depicted in Figure 1.

In the system abstraction routine, an abstraction framework is created that allows for numerical training via reinforcement learning techniques and abstraction based verification. In this routine the neural network is also formatted to conform to the abstraction framework. After the abstraction framework is setup the actual synthesis begins.

The synthesis procedure switches between the network training routine and the verification routine. During the network training routine, the neural network controller is training using reinforcement learning techniques based on the performance of finite episodes. These episodes are simulations of the interaction between the neural network controller and the simulated plant. This causes the neural network to learn to control the plant appropriately.

After an arbitrary amount of training the procedure switches to the system verification routine. During the verification routine the abstraction framework is exploited to create a partial abstraction that simulates the original system. This is an abstraction of the neural network feedback controlled system. The verification routine then finds the guarantees that the neural network controller boasts using fixed-point algorithms. If the resulting guarantees are deemed sufficient the synthesis procedure is ended. Otherwise, the resulting guarantees can be used to refine the network training routine. COSYNNC currently supports the invariance, reachability and reach and stay control specification.

3 Technical details

In this section a more in depth overview of the functioning of the framework will be provided. First the neural network controller topology will be elaborated upon. After that, the three routines that make up the synthesis procedure will be presented in more technical detail.

3.1 Neural network controller topology

For a neural network to function as a controller, it requires a topology such that it can take a representation of the state of the plant as an input and output an input for the plant. It should do so such that a control specification is adhered. For this purpose, COSYNNC ships with two neural network controller topologies by default: labelled and unlabelled output neurons. In both topologies, a quantized and normalized

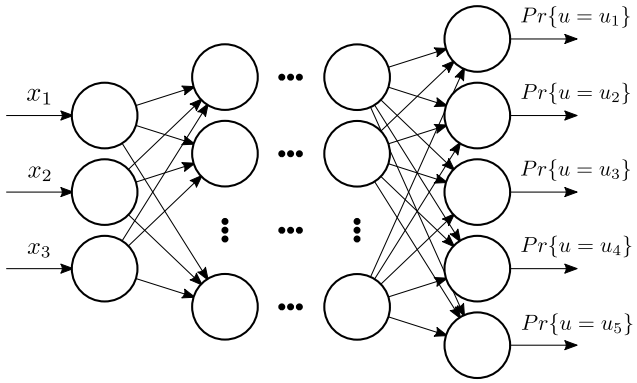

Figure 2: A graphical representation of a multilayer perceptron neural network controller. This particular topology uses a normalized quantized representation the state of the plant as input to the neural network and labelled output neurons to represent the input to the plant.

representation of the state space is fed to the neural network controller. This is achieved by dedicating an input neuron of the neural network to each dimension of the plant and feeding it the quantized and normalized state of the plant.

In the labelled topology's case, each of the inputs in the finite cardinality input space of the abstraction is given a dedicated output neuron of the neural network. The neural network output hence becomes a probabilistic representation of the input to the plant where each output neuron represents the chance that its labelled input is the appropriate one. A figure representing this neural network controller topology for a multilayer perceptron neural network is depicted in Figure 2.

Alternatively, an unlabelled topology can be used. In this topology, each dimension in the input space is represented by merely two output neurons. These output neurons determine the range in which any probabilistic input to the plant will be. The probability is uniformly distributed over all the inputs in that range. This has the advantage of not being limited by the cardinality of the input space. In practice, this particular topology does however behave more erratically and is more difficult to train successfully.

3.2 System abstraction

The system abstraction routine is the initial routine of the synthesis procedure and is responsible for setting up a simulating abstraction framework for the system of interest. Creating such a simulating abstraction framework with a finite cardinality input and state space allows for the use of numerical methods. This allows for both the training of the neural network controller and more importantly the verification of the neural network controller. In this framework, a state quantizer is added in front of the neural network controller and input quantizer is added in front of the plant. By introducing these quantizers, the finite cardinality of the state and input space that is seen by the plant is guaranteed. The simulating relation that is used by the abstraction is the feedback refinement relation [2].

3.3 Network training

During the training routine, the procedure switches between running training episodes and performing reinforcement learning based on the performance of those episodes to train the neural network controller.

To train the neural network controller, reinforcement learning techniques [3–5] are used to train the controller based on the performance of episodes. An episode is a finite horizon iteration of the system in closed loop with the neural network acting as the controller. Each episode starts by picking an initial state from the initial state set I_0. During an episode, the state of the plant is fed through the state quantizer, normalized and provided to the neural network. The neural network will then give a probabilistic output which is the input of the plant after being fed through the input quantizer. The plant is then iterated by sampling an input and the cycle repeated. This particular feedback loop is iterated until a termination condition is met.

The termination condition is met whenever the episode reaches its finite horizon $i > N$. Another obvious termination condition is whenever the state of the plant leaves the bounds of the restricted state space. In addition to these conditions, other conditions can be used based on the control specification to speed up the synthesis procedure. For example, in the case of a reachability control specification, it suffices to terminate the episode once the to be reached set has been reached.

Once the episode is terminated, the performance of the episode is evaluated and based on the performance the neural network controller is reinforced or deterred using reinforcement learning methods. The metric through which the performance of the episode is evaluated is based on the control specification. The performance indicator that COSYNNC uses is whether or not the episode adhered to the control specification on the finite horizon. If that is the case the controller's policy is reinforced, if not its policy is deterred.

To reinforce or deter the control policy of the neural network controller, so called policy gradients need to be computed. These policy gradients provide the gradients for the weights and biases such that the input probabilities are tweaked based on the performance metric. An elementary algorithm to calculate these policy gradients for a labelled input neural network controller is provided 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 list of normalized quantized states in the episode and $\mathcal{U}$ is the list of inputs in the episode. The function $f(\vec{x}, \boldsymbol{\theta})$ represents the neural network's output as a function of its input $\vec{x}$ and its parameters $\boldsymbol{\theta}$. The function $i : \mathbb{N}_0 \to \mathbb{R}^n$ maps the index j of the output neurons to the actual input which that neuron labels.

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

procedure PolicyGradient($\mathcal{P}$, $\mathcal{X}$, $\mathcal{U}$)
  $\mathcal{L} \leftarrow 0$
  for $i \leftarrow 1, N$ **do**
    $\vec{x} \leftarrow \mathcal{X}_i$
    $\vec{u} \leftarrow \mathcal{U}_i$
    $\vec{u}^{nn} \leftarrow f(\vec{x}, \boldsymbol{\theta})$
    for $j \leftarrow 1, |\vec{u}|$ **do**
      if $\mathcal{P} = 1$ **then**
        if $\vec{u} = i(j)$ **then**
          $L_j \leftarrow 1$
        else
          $L_j \leftarrow 0$
        end if
      else
        if $\vec{u} = i(j)$ **then**
          $L_j \leftarrow 0$
        else
          $L_j \leftarrow 1$
        end if
      end if
    end for
    $\mathcal{L} \leftarrow \mathcal{L} - \sum_{j=1}^{|\vec{u}|} \|L_j\| \log(u_j^{nn})$
  end for
  $\theta \leftarrow \theta - \lambda \nabla_\theta \mathcal{L}$
end procedure

3.4 System verification

During the system verification routine, the set for which the neural network controller currently adheres to the control specification, called the winning set W, is to be calculated. To do this, during each verification routine, a partial abstraction is generated based on the current neural network controller. A fixed-point

algorithm is then performed to find the winning set W. Based on the winning set W, finally synthesis guiding is applied to refine the training routine.

3.4.1 Partial abstraction

Once an arbitrary number of training episodes have refined the neural network controller the synthesis procedure is switched to the verification routine. In the verification routine the system abstraction framework is used to generate a partial abstraction. This partial abstraction S_p is a simulating system of finite state cardinality. In this partial abstraction system, the state space of the partial abstraction is a cover of the state space based on the bounds and quantization parameter of the state quantizer. The input space is similarly determined by the input quantizer. The resulting states are thus rectangular hyper-cells that cover the state space. The transitions between these hyper-cells are then determined by the greedily picking the input that the neural network controller outputs for that state. The resulting transition function for the abstraction thus becomes the set of hyper-cells that intersect with the over-approximation of the dynamics of all the hyper-cells. This result is a partial abstraction S_p that simulates the system S and can thus be used for verification. Mathematically this partial abstraction S_p can be described using Reissig's [2] feedback refinement relation as:

- $\bar{X}_p$ 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.
- $U_p \subseteq U \mid \forall u_p \in U_p \; \exists \vec{c} \in \bar{X}_p$ s.t. $f(\vec{c}, \boldsymbol{\theta}) \implies u_p$
- $(x_p, u_p, x'_p) \in \rightarrow \mid x_p \in \bar{X}_p, u_p = f(\vec{c}, \boldsymbol{\theta}) \in U_p$ and $x'_p \in \bar{X}_p$ s.t. $\text{Post}_{u_p}(x_p) \cap x'_p \neq \emptyset$
- $\text{Post}_{u_p}(x_p) = \emptyset$ whenever $x_p \in X \setminus \bar{X}_p, u_p \in U_p$

where $f(\vec{c}, \boldsymbol{\theta})$ denotes the greedy result from the neural network controller, $\vec{c}$ is the center of the hyper-cell and $\text{Post}_{u_p}(x_p)$ is a function that returns the set to which the dynamics of the plant map based on the initial cell and the input.

To use this abstraction numerically, the transfer function $\text{Post}_{u_p}(x_p)$ needs to be defined that returns the exact or otherwise over-approximated set of states to which the initial hyper-cell maps based on the input. In this paper, two types of over-approximations are used for this purpose. The type that is used is based on whether or not the dynamics of the plant classify as linear dynamics or nonlinear dynamics.

In case of linear dynamics, the evolution of a hyper-cell can be found by considering the vertices of the hyper-cell. Iterating the vertices of the hyper-cells with the dynamics of the plant yields a convex hull that represents the set into which any state in the original hyper-cell will map. The transfer function for any state thus becomes the set of states that intersect the convex hull that results from iterating the vertices of the hyper-cell that contains that state.

For nonlinear dynamics, the over-approximation of the hyper-cell is based on a radial growth bound function $\beta(r, \vec{u})$ where r is the radius of the hyper-cell and $\vec{u}$ the input for that cell. This function provides a radial bound on the evolution of the hyper-cell based on the nonlinear dynamics.

3.4.2 Fixed-point iteration

Based on the resulting over-approximated partial abstraction, the behaviour of the closed loop system can now be verified. For this purpose, fixed-point algorithms are employed. Running such a set-based fixed-point algorithm until a fixed-point is reached will result in the subset of states $W \subseteq \bar{X}_p$, coined the winning set, for which the neural network controller is able to adhere to the control specification. This particular methodology therefore results in a neural network controller with formal guarantees on its behaviour.

3.4.3 Synthesis guiding

Once the winning set W is calculated, it can be used during the training routine to refine the training focus. By doing so the synthesis procedure can be sped up by focusing on problematic states in $\bar{X}_p$. The training focus is set by changing the initial state set for the episodes I_0 to a subset of the state space based on the winning set W.

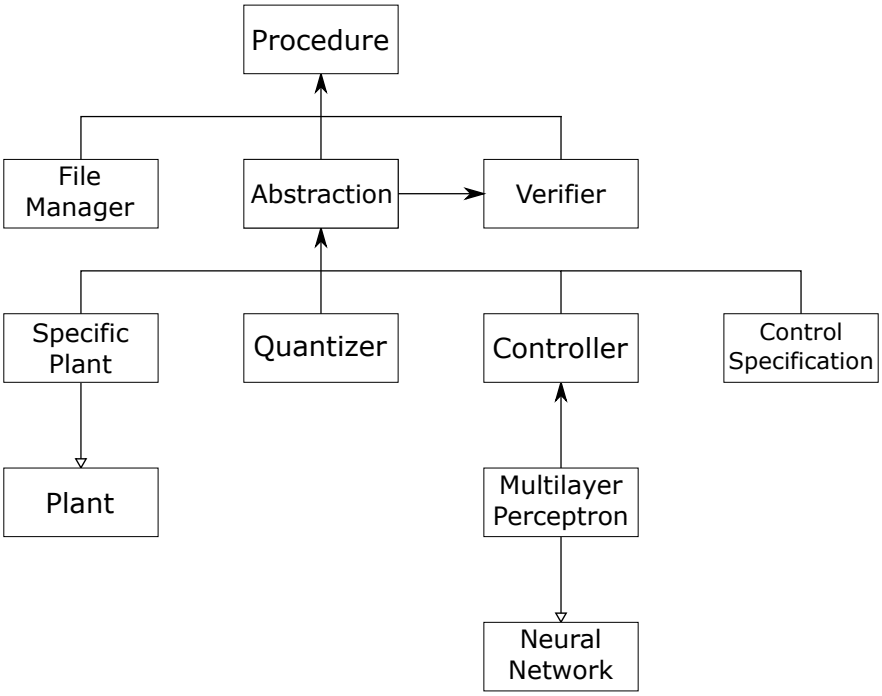

Figure 3: An overview of the structure of the COSYNNC framework. The filled arrows indicate that that class is instantiated or used by the class towards which it points. The hollow arrows means that the class from whence that arrow came inherits its property and behaviour from the class towards which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

A wide variety of options are available to define this set and often a combination of various methods is used. Amongst which is to set $I_0 = W^c$ such that the neural network controller is forced to train on episodes that start in the losing set. In this way the controller focuses on problematic states and the chance of finding the appropriate inputs (if they exist) is increased.

In practice, combinations of different training focuses have been used. Primarily, the episode initial state set I_0 is switched between containing all the states in the space, the complement of the winning set W^c and a set of states that radially growth with respect to the control goal set. Using this combination of focuses prevents the the neural network controller from 'over training' on a certain set and losing previously acquired behaviour.

4 Framework usage

In this section, more details with regards to using the framework shall be provided.

The current implementation of the framework is written using C++14 standard. The framework uses the MXNet deep learning library[1] for all of the native neural network manipulations. An overview of the structure of the framework is depicted in Figure 3. The framework can be downloaded from https://github.com/WardvanderVelden/COSYNNC.

In order to perform correct-by-design neural network control synthesis using the standard COSYNNC framework the user has provide:

- A representation of the plant in the form of a class inheriting from the generic *Plant* class
- The neural network topology onto which the controller should be encoded
- The parameters for the abstraction framework

[1]The MXNet deep learning library can be found at: https://mxnet.apache.org/

- The parameters for the neural network training

First a representation of the plant is required. This is a C++14 class that inherits from the generic *Plant* class that COSYNNC provides. This representation includes the dynamics of the plant, the over-approximation of the plant in case of a nonlinear plant and the sampling time and dimensions of the plant. An example of such a representation of the plant for a nonlinear unicycle system is provided in Section 4.1.

Once the plant is defined the other synthesis parameters can be specified in order to execute the synthesis procedure. First the neural network controller needs to be defined. The procedure need to be provided with a neural network object that inherits from the *NeuralNetwork* class. COSYNNC is shipped with a *MultilayerPerceptron* class that adheres to this requirement and embeds the behaviour of a multilayer perceptron neural network. The user has to specify the number of layers and neurons in each layer of the neural network. It also has to specify the controller topology the neural network will comply to. This determines the amount of input and output neurons the neural network will have. Standard COSYNNC includes the *Labelled* and *Unlabelled* neural network controller topology. In addition to this, the user also has to specify the nonlinear activation function used by the neural network. Finally, the user has to specify the learning rate λ and the backpropagation algorithm. For the available activation functions and backpropagation algorithms the reader is referred to the MXNet documentation[2].

After defining the neural network, the general synthesis parameters can be specified. This includes defining the state and input quantizer parameters which in turn define the partial abstractions. This also includes the control specification for which the neural network should train. COSYNNC ships with the *Invariance*, *Reachability* and *ReachAndStay* specifications. Finally the training focuses can be defined which speed up the synthesis procedure by integrating the system verification and neural network training routine. The training focuses that the current implementation of COSYNNC has are: *SingleState*, *AllStates*, *RadialOutwards*, *LosingStates* and *NeighboringLosingStates*. Finally the user also has the option to enable some experimental synthesis options such as *WinningSetReinforcement* and *NormReinforcement*.

COSYNNC has a number of different outputs available by default. These outputs and their meanings are listed in Table 1.

Table 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

COSYNNC is also shipped with a number of tools that allow the user to convert these representations into other representations for data comparison purposes. These tools are mainly intended to be used as an inferface with the correct-by-design synthesis tool SCOTS [6]. By using these utilitarian tools, users are able to make their own comparison with different correct-by-design synthesis tools to see how they compare to COSYNNC's neural network controllers.

COSYNNC is intended as a framework for correct-by-design neural network controller synthesis. As such,

[2]The C++ MXNet documentation reference: https://mxnet.apache.org/versions/1.6/api/cpp/docs/api/index.html

users are encouraged to make changes, add other neural network topologies and in general use the framework to further research into these types of controller.

4.1 Example code

An example of the code used to evoke the synthesis procedure for the unicycle system is:

Unicycle.h

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

namespace COSYNNC {
  class Unicycle : public Plant {
  public:
    Unicycle() : Plant(3, 1, 0.3, "Unicycle", false) { };

    Vector DynamicsODE(Vector x, float t) override;
    Vector RadialGrowthBoundODE(Vector r, float t) override;
  private:
    double _v = 1.0;
    double _omega = 2.0;
  };
}
```

Unicycle.cpp

```
#include "Unicycle.h"

namespace COSYNNC {
  Vector Unicycle::DynamicsODE(Vector x, float t) {
    Vector dxdt(_stateSpaceDimension);
    dxdt[0] = _v * cos(x[2]);
    dxdt[1] = _v * sin(x[2]);
    dxdt[2] = _u[0] * _omega;

    return dxdt;
  }

  Vector Unicycle::RadialGrowthBoundODE(Vector r, float t) {
    Vector drdt(_stateSpaceDimension);
    drdt[0] = r[2] * _v;
    drdt[1] = r[2] * _v;
    drdt[2] = 0;

    return drdt;
  }
}
```

Inside *main.cpp*

```
Unicycle* plant = new Unicycle();
// Define the neural network
MultilayerPerceptron* mlp = new MultilayerPerceptron({ 12, 12 }, ActivationActType::kRelu, OutputType::
    Labelled);
mlp->InitializeOptimizer("sgd", 0.0075, 0.0);
// Initialize the procedure and set the plant
Procedure cosynnc;
cosynnc.SetPlant(plant);
// Specify the quantizer parameters
cosynnc.SpecifyStateQuantizer(Vector({ 0.1, 0.1, 0.1 }), Vector({ -5.0, -5.0, 0 }), Vector({ 5.0, 5.0, 2
    * PI }));
cosynnc.SpecifyInputQuantizer(Vector({ 1.0 }), Vector({ -1.0 }), Vector({ 1.0 }));
// Specify the neural network training parameters
cosynnc.SpecifySynthesisParameters(5000000, 50, 5000, 50000, 100);
cosynnc.SetNeuralNetwork(mlp, 5);
// Specify the control specification
cosynnc.SpecifyControlSpecification(ControlSpecificationType::Reachability, Vector({ -2.05, -2.05, 0 }),
    Vector({ 2.05, 2.05, 2*PI }));
// Specify the training focuses
cosynnc.SpecifyTrainingFocus(TrainingFocus::RadialOutwards);
cosynnc.SpecifyRadialInitialState(0.5, 1.0);
cosynnc.SpecifyTrainingFocus(TrainingFocus::NeighboringLosingStates);
cosynnc.SpecifyTrainingFocus(TrainingFocus::AllStates);
// Specify if the refined transitions are used
cosynnc.SpecifyUseRefinedTransitions(true);
// Specify where the controllers are saved
cosynnc.SpecifySavingPath("../controllers");
// Initialize the entire procedure and synthesize
cosynnc.Initialize();
cosynnc.Synthesize();
// Free up memory
delete mlp;
delete plant;
```

5 Function overview

In this section an overview of the main callable functions in the procedure and encoder class shall be given together with a small description. This overview serves to provide the user with some insights with regards to the different synthesis parameters that are available within COSYNNC.

For *Procedure.h*:

Function name	Function description
SetPlant	Sets the plant.
SetNeuralNetwork	Sets the neural network.
SpecifyStateQuantizer	Specify the state quantizer to the specified quantization parameters.
SpecifyInputQuantizer	Specify the input quantizer to the specified quantization parameters.
SpecifyControlSpecification	Specify the control specification.
SpecifySynthesisParameters	Specify the control synthesis parameters including max episodes, episode horizon and verification episode.
SpecifyTrainingFocus	Specify the training focus that should be used during training, this will then be added to the list of training focuses.
SpecifyRadialInitialState	Specify the radial initial state selection based on the synthesis progression.
SpecifyNorm	Specify the norm to be used during training.
SpecifyVerbosity	Specify the verbosity of the procedure.
SpecifyWinningSetReinforcement	Specify if the network should reinforce upon reaching the winning set (only applicable to reachability).
SpecifySaveRawNeuralNetwork	Specify if the raw neural network should be saved.

SpecifyComputeApparentWinningSet	Specify if the apparent winning set should be computed. This is the winning set from which the center of each hyper-cell adheres to the control specification.
SpecifyUseRefinedTransitions	pecify what type of transition calculation should be used.
SpecifySaveAbstractionTransitions	Specify whether or not to save the transitions.
SpecifySavingPath	Specify the saving path.
Initialize	Initialize the procedure, returns false if not all required parameters are specified.
Synthesize	Run the synthesis procedure.
Verify	Run the verification phase.
LoadNeuralNetwork	Load a neural network.
SaveNetwork	Save the neural network.
SaveTimestampedNetwork	Save the neural network as a timestamp.

For *Encoder.h*:

Function name	Function description
SetNeuralNetwork	Set the neural network.
SetBatchSize	Set the batch size of the encoder during training.
SetThreshold	Set the threshold that determines the significance with which the neural network needs to output a value for it to be taken as the truth.
SetSavingPath	Set the saving path to which the neural networks save.
Train	Train the linked neural network to encode the winning set.
ComputeFitness	Compute the fitness of the neural network to determine what part is currently encoded succesfully.
ComputeFalsePositives	Compute the amount of false positivies that the neural network provides with respect to the cardinality of the state space.
DetrainFalsePositives	Detrain the false positives that are currently in the network.
Encode	Encodes the winning set of the loaded static controller into the neural network.

References

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

[2] G. Reissig, A. Weber, and M. Rungger, "Feedback refinement relations for the synthesis of symbolic controllers," *IEEE Transactions on Automatic Control*, vol. 62, pp. 1781–1796, April 2017.

[3] L. P. Kaelbling, M. L. Littman, and A. W. Moore, "Reinforcement learning: A survey," *Journal of Artificial Intelligence Research*, vol. 4, pp. 237–285, 1996.

[4] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Machine Learning*, pp. 229–256, 1992.

[5] A. Karpathy, "Deep reinforcement learning: Pong from pixels," May 2016.

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