ABSTRACT

Title of thesis: EXPANDING CONSTRAINED KINODYNAMIC PATH PLANNING SOLUTIONS THROUGH RECURRENT NEURAL NETWORKS

Joshua Shaffer
Master of Science, 2019

Thesis directed by: Dr. Huan Xu
Department of Aerospace Engineering

Path planning for autonomous systems with the inclusion of environment and kinematic/dynamic constraints encompasses a broad range of methodologies, often providing trade-offs between computation speed and variety/types of constraints satisfied. Therefore, an approach that can incorporate full kinematics/dynamics and environment constraints alongside greater computation speeds is of great interest. This thesis explores a methodology for using a slower-speed, robust kinematic/dynamic path planner for generating state path solutions, from which a recurrent neural network is trained upon. This path planning recurrent neural network is then used to generate state paths that a path-tracking controller can follow, trending the desired optimal solution. Improvements are made to the use of a kinodynamic rapidly-exploring random tree and a whole-path reinforcement training scheme for use in the methodology. Applications to 3 scenarios, including obstacle avoidance with 2D dynamics, 10-agent synchronized rendezvous with 2D dynamics, and a fully actuated double pendulum, illustrate the desired performance

of the methodology while also pointing out the need for stronger training and amounts of training data. Last, a bounded set propagation algorithm is improved to provide the initial steps for formally verifying state paths produced by the path planning recurrent neural network.

EXPANDING CONSTRAINED KINODYNAMIC PATH PLANNING SOLUTIONS THROUGH RECURRENT NEURAL NETWORKS

by

Joshua Shaffer

Thesis submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Master of Science
2019

Advisory Committee:
Dr. Huan Xu, Chair/Advisor
Dr. Michael Otte
Dr. Derek Paley

Acknowledgments

I would first like to acknowledge and thank my advisor, Dr. Huan Xu, in the Department of Aerospace Engineering at the University of Maryland. Thanks to her support and guidance, I was able to explore a broad range of interesting and important research topics throughout my time with the Aerospace Engineering Department, all of which helped build towards this body of work. I am greatly appreciative of her patience and knowledge as I worked towards this thesis. Second, I would like to thank the staff and faculty of the Aerospace Engineering Department for their provided resources and knowledge throughout my two years at the University of Maryland. I am proud to have worked and learned as part of such a strong graduate program. Last, I would like to thank my family and friends throughout this experience. My parents specifically, Lisa and Jeffrey Shaffer, have always strongly supported my work and endeavors, and I am grateful to be their son.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

BPTT	Backpropagation through time
BVP	Boundary value problem
GRU	Gated Recurrent Unit
LSTM	Long Short-term Memory
ML	Machine learning
MLP	Multilayer perceptron
NARX	Nonlinear Autoregressive Network with Exogenous Inputs
NLP	Nonlinear programming
NN	Neural network
RL	Reinforcement learning
RMSE	Root mean square error
RNN	Recurrent neural network
RRT	Rapidly-exploring Random Tree
RRT*	Optimal Rapidly-exploring Random Tree
SGD	Stochastic gradient descent
SQP	Sequential quadratic programming

Chapter 1: Introduction

1.1 Motivation

Path planning for general autonomous vehicles encompasses a broad range of methodologies, from which the resulting applications require varying degrees of accuracy and computational speed for the solutions solved. The introduction of kinematic/dynamic constraints alongside optimality conditions (often called motion or trajectory planning) increases the difficulty in producing quick and feasible solutions [1]. As a result, faster path planning methodologies tend to ignore or greatly reduce kinematic/dynamic and optimality constraints in favor of finding solutions to satisfy constraints tied to a varied environment [2], [3]. Often times in these approaches when kinematics/dynamics and optimality conditions must be considered, they are abstracted to simpler models alongside the use of heuristics, from which controllers must enforce in real-time [4]. Application performance of such methods is greatly dependent upon how well the abstracted system applies to the full dynamic model and controller utilized.

In the cases where the total solution must not only consider environmental constraints but also full dynamic/kinematic constraints alongside optimality conditions, formulating the problem as a trajectory optimization problem and solving such numerically can provide the desired results. Typically, though, direct optimization methods (which

discretize the state and control path as a function of time before optimizing) involve higher computation times than path planning methods that ignore or reduce constraints and optimality conditions [5]. This impacts the real-time performance of trajectory optimization schemes [4], from which increased performance typically requires some form of problem relaxation [6]. Furthermore, increases in state space size only exacerbate the issue [7], [1], [4].

Because of the accuracy and computation speed trade-off between these various approaches to path planning with constraints, an encompassing solution that can incorporate kinematics/dynamics and greater environment constraints in its formulation alongside the speed of methods that reduce these constraints to simpler or non-existent models is of great interest. Decreasing the computation time associated with an algorithm that achieves such a goal typically requires some approach of providing strong initial guesses to speed up generalized methods [8], or re-planning from adequate (but quickly obtained) initial guesses [9]. At the core of these examples is the notion that speeding up complex path generation requires quick initial guesses that may or may not be the best solution to the desired problem. This idea motivates this thesis's exploration of machine learning (specifically the use of neural networks (NNs)) in adapting specific optimization solutions to generate solutions to a general workspace.

Supervised machine learning with recurrent neural networks (RNNs) provides a platform from which unknown time-dependent processes can be form-fitted through training data in which the correct input and output sets are known. In general, RNNs differ from the standard feedforward NNs in that states of the networks are maintained throughout execution, resulting in outputs that are informed by previous inputs. This attribute

enables RNNs to better learn time-dependent processes, such as mapping disease descriptions mentioned sequentially in text [10] or predicting radiation fluctuations due to weather changes [11]. Various forms of RNNs exist, such as nonlinear autoregressive with exogenous inputs (NARX), gated recurrent unit (GRUs), and long short-term memory (LSTM) NNs, of which no single network is best suited to learn any general time-depended behavior [12]. In general, the benefits of RNNs (and NNs overall) are their computational speed in providing results, ability to abstract complex systems, and ability to generalize complex solutions for use with new inputs [13].

On the flip-side though, the use of NNs impose two major limitations. First, large amounts of data are required to train networks for complex problems in a supervised manner, an issue when considering the time required to generate sets of optimized solutions to kinematic/dynamic path planning scenarios. Second, NNs create a major hurdle with respect to obtaining verifiable results due to their often treatment as black box solutions [13], a detriment in the areas of path planning and control of kinematic/dynamic systems. Despite such, methods exist to verify NNs, and compose a growing field in pursuit of verifying the use of NNs for safety critical applications [14], [15], [16].

The use of NNs, specifically RNNs, in learning optimized solutions to constrained kinematic/dynamic path planning solutions is a relatively unexplored area of research (especially on continuous domains), in part due to the issues of verification. The benefits of using RNNs (generalizing solutions to new environments and requiring extremely low computation costs) provide enough motivation to explore their applications in path planning. Specifically, this thesis explores a generalized methodology for obtaining optimized path planning results as training data, training RNNs to recreate such results, assessing

controlled paths over the generated paths. Additionally, a method of verifying RNN outputs for a large set of possible inputs is also explored.

1.2 Related Work

Various sources have explored the uses of NNs in controls and path planning of kinematic/dynamic systems. This is a growing field of interest, primarily due to the ability of NNs in generalizing solutions to new environments (beneficial to path planners) and abstracting complex systems (beneficial to control systems).

In relation to controls, RNNs often find uses as controllers of complex systems, e.g. in highly nonlinear systems [17], [18]. In some cases, the use of learning in controllers is well defined in order to obtain greater understanding of their effects on system stability and convergence [19]. Additional uses of machine learning in the controls community are focused towards creating models of complex, unknown dynamics [20], [21]. These various examples showcase the ability of various RNN architectures in abstracting and/or identifying complex kinematic/dynamic system relationships, an important aspect with respect to the aims of this thesis.

The use of machine learning directly in relation to path planning of kinematic/dynamic systems is an expansive research topic, in part due to the opposing nature between the need for constraint satisfaction in generated paths and the difficulty in formally verifying machine learning outputs [22]. Despite such hurdles, various authors have investigated the benefits of machine learning in aiding path planning, albeit from varying angles.

[23] demonstrates one of the earliest cases in using NNs to generate feasible paths

in an environment with dynamic obstacles. Specifically, a 2-dimensional region was decomposed into discrete units from which each was represented by a neuron, with output connections to adjacent neurons and an output to represent path feasibility. This approach, while fast in generating a path, primarily minimized the distance traveled and simply avoided obstacles. Additionally, no dynamic constraints were considered, and an abstraction of the state space had to be used. This approach was similarly used in [24]. Along the same vein, [25] utilized pulse-coupled NNs for determining the shortest path in an unknown environment, similarly to [23]. Again, the state space was discretized, and dynamics were enforced at lower levels. [25] also performed physical tests, though, and moved towards validating the practical application of a NNs based approach.

[26] examined the specific application of using a 2-input, 2-output NN to provide controls to an interplanetary rover navigating a known terrain, tested in simulation. Specifically, the 2 inputs represented x and y coordinates, with the outputs representing control signals to the wheels. In this case, training was performed on a static environment, and the rover had to navigate from any point to a static final destination. This work showcases the ability for a simpler network to accurately abstract the dynamics and controls associated with using position feedback to drive a robot to a final destination, a concept also successfully explored in [27]. Additionally, [28] achieved such an approach formulated in a local frame about the robot for easy integration with actual sensor data, achieving practical application. As opposed to the controls-focused sources that explored the ability of NNs to abstract kinematic/dynamic systems, these NN applications abstracted the kinematic/dynamic systems through environment feedback, i.e. NN outputs provided references to lower level controllers which modified the system state and,

in effect, the environment. As a result, the controllers (created to track a reference signal) were abstracted into the environment for use by the NN.

Interestingly, [26] , [27], and [28] represent path planning approaches related to a more recently growing field in machine learning called deep reinforcement learning (RL). Deep RL utilizes deep NNs as a policy that maps perception inputs to outputs, from which the NN is trained to provide actions that maximize some reward. The benefit of such an approach is that complex perception inputs (e.g. camera data) can provide action outputs, such as camera data fed into a robot to produce velocity controls as shown in [28]. More examples include the use of LIDAR data to drive local position commands in [29] and the use of local visual data to force drone agents to produce flocking velocity commands amongst a group of agents in [30]. In most deep RL applications, training is performed utilizing agent outputs only (RL only) or agent outputs compared to desired strategies or models (RL with supervised training). In the case of RL with supervised training, no sources could be found in which the model for a path planning RNN consisted of optimized trajectory data (i.e. training examples contained the sequence of actions that produce the maximized reward used in reinforced learning for the NN). For this thesis, the purpose of RL is to showcases how a sequence of results produced by a NN can be improved through training the network utilizing these results. This idea is used in improving the overall methodology performance presented in this thesis.

The authors in [8] and [31] present methodologies most similar to the one presented in this thesis. In [8], regression learning is utilized to select previous path planner solutions for a robotic manipulator as initial guesses for the optimization planner, resulting in speed-ups of up to an order in magnitude. In [31], an RNN is utilized to learn from

shortest path solutions generated by an RRT between two points provided randomized obstacles. Additionally, an environment autoencoder network is utilized to reduce the state space size of the environment and robustly represent such constraints. The result is an RNN that takes in any obstacle configuration and produces the shortest path between a current location and the desired final location. The computational advantages were up to an order of magnitude or more when compared to some of the fastest conventional planners, and scaled well with larger state spaces.

Each of the aforementioned approaches represent varying ways of utilizing networks in path planning, with the foremost advantage of decreased computation costs in execution. With respect to our work, [26], [27], and [28] showcased the ability of a network to abstract dynamics of specific scenarios under environment inputs, and [8] and [31] displayed the ability of a network to generate state paths with respect to a specific optimization parameter for highly varied environments. What is lacking from existing literature is an approach of utilizing known optimization solutions for generalized kinematic/dynamic path planning and embedding the solution space directly into an RNN to generate desired state paths through time. The methodology presented in this thesis addresses this gap by integrating an RNN's abilities of learning kinematic/dynamics and generating optimized paths through reinforcement training upon previously optimized solutions.

1.3 Contributions

The contributions of this thesis include three primary components, two of which are integrated into the overall methodology presented. First, this thesis contributes a means of utilizing non-differentially flat systems in optimal path planning with rapidly-expanding random trees (RRT). This contribution is achieved by extending the formulation of a kino-dynamic RRT from utilizing B-spline representations that require differentially flat systems to utilizing collocation methods that allow for generic nonlinear systems and control solutions. This allows for a broader range of problem scenarios from which optimization solutions can be obtained.

Second, this thesis contributes a means of improving closed-loop executions of RNNs in predicting state paths for path planning purposes. This is achieved though a multi-step training scheme that first utilizes supervised training of the RNN on desired paths then incorporates reinforcement learning ideas to iteratively train the network through its closed-loop outputs. This improvement reduces prediction errors of the RNN as it generates the paths over time and allows it to better fit the optimization solution space.

Third, this thesis contributes a means of speeding up and improving the accuracy of bounded set propagation of RNN states for verification purposes. This is achieved through a bisection algorithm over the propagated hyperrectangles, which produces more accurate results than previously researched methods. Bounded set propagation is a vital tool in reachability analysis, allowing for verification that RNN paths generated from regions of initial conditions in set environments satisfy desired constraints.

Last, this thesis contributes a methodology for training an RNN to produce state paths from optimized kinematic/dynamic formulations and building controller solutions on top of such, the result of which is a computationally fast path planner and control approach. This is achieved through integration of the first and second contributions in order to generate supervised data from which a path-planning RNN is trained upon. Applications of this methodology in this thesis provide outlooks on how path-tracking controllers can be built on top of the generated paths. Compared to some of the bigger advances of motion controls in the deep RL field, this methodology provides a more segmented approach since the RNN only produces state paths in time, from which controls are constructed on top of. The aim of this approach is to provide an easier path towards formally verifying the combined solutions, since separate verification methods exist for predicting RNN outputs and for predicting state paths of formally defined control laws.

1.4 Outline of Thesis

The outline of the rest of this thesis is as follows. Chapter 2 introduces background material on the components of the methodology, including details pertaining path planning, machine learning, and formal verification methods. Chapter 3 introduces the methodology and formal problem description of its components. Chapter 4 provides details on the improvements to the RRT and training scheme used for the methodology. Chapter 5, 6, and 7 each present an application of the methodology alongside results. Chapter 8 presents the improvements on the bounded set propagation method, and Chapter 9 concludes the thesis.

Chapter 2: Background

Chapter 2 introduces the base concepts explored and expanded upon in this thesis. These include the use of RRTs in path and motion planning, the training and use of RNNs in machine learning of time-dependent series, and bounded-set propagation for formal verification.

2.1 Path Planning

Path planning subject to kinematic/dynamic constraints constitutes the objective of creating a state and control trajectory from an initial state configuration to a final configuration under added environment constraints and optimization metrics (composed of functions on state, control, and time). Methods proposed across engineering disciplines for solving path planning problems typically fall into categories based on the desired constraints to satisfy. As mentioned in Chapter 1.1 of this thesis, encompassing solutions that include environment constraints (such as obstacles) alongside kinematic/dynamic constraints typically require greater computation times than methods focused on satisfying just environments or just kinematics/dynamics. Furthermore, optimization approaches that can solve for both sets of constraints typically require modifications to one set or the other. For example, the satisfaction of logic constraints (e.g. preventing a state from

entering an arbitrarily shaped region) requires convex reshaping of such constraints for uses in convex optimization formulations. This type of scenario showcases not only the difficulties in obtaining path planning solutions for a broad set of environment and kinematic/dynamic constraints, but also obtaining such in reasonable time frames.

One set of approaches for obtaining optimal path planning solutions under arbitrary environment constraints include sampling-based techniques. Sampling-based methods utilize random state sampling to construct paths from an initial condition to an end condition, removing any possible paths that violate constraints. Further refinements over time eventually build towards the optimal solution (with respect to the defined metric function). The trade-off, though, is the time required to find the optimal solution, as most sampling-methods are only complete in a probabilistic sense as the number of samples approaches infinity. One of the most prevalent sampling methods is the optimal rapidly-exploring random tree (RRT*), presented in the following section.

[Note: The introduction of kinematic/dynamic constraints alongside optimization over functions of the path variables to path planning equates it to motion/trajectory planning, and references to any of the three terms in this thesis constitute the same idea.]

2.1.1 Optimal Rapidly-exploring Random Trees (RRT*)

Optimal rapidly-exploring random trees (RRT*) introduced in [32] present a method of state sampling to build an optimal tree (shortest state path distance) from an initial configuration to a final configuration. Provided an initial state x_{init}, logical constraint functions $Check\_constraint(path)$ over discrete values in a path, cost function $Cost(x_1, x_2)$

between two points, and graph G composed of state nodes V and edges E, RRT* operates as described in Algorithm 1.

Algorithm 1: Optimal Rapidly-exploring Random Tree from [32]

procedure GENERATE_PATH(x_{init}, N) ▷ Execute RRT* over N iterations
  $V \leftarrow \{x_{init}\}$; $E \leftarrow \emptyset$
  for $i = 1, ..., n$ **do**
    $x_{rand} \leftarrow$ Sample state
    $x_{nearest} \leftarrow \text{argmin}_{v \in V} \|x_{rand} - v\|$
    $x_{new} \leftarrow \text{argmin}_{\|z - x_{nearest}\| \leq \eta} \|z - x_{rand}\|$
    if $Check\_constraint(path(x_{nearest}, x_{new}))$ **then**
      $X_{near} \leftarrow \{v \in V \mid \|x_{new} - v\| \leq radius\}$
      $V \leftarrow V \cup \{x_{new}\}$
      $x_{min} \leftarrow x_{nearest}$; $c_{min} \leftarrow \sum_G Cost(x_{nearest}) + Cost(x_{nearest}, x_{new})$
      for $x_{near} \in X_{near}$ **do**
        if $Check\_constraint(path(x_{near}, x_{new})) \wedge \sum_G Cost(x_{near}) + Cost(x_{near}, x_{new}) < c_{min}$ **then** ▷ Determine minimum cost connection
          $x_{min} \leftarrow x_{near}$; $c_{min} \leftarrow \sum_G Cost(x_{near}) + Cost(x_{near}, x_{new})$
      $E \leftarrow E \cup \{(x_{min}, x_{new})\}$
      for $x_{near} \in X_{near}$ **do**
        if $Check\_constraint(path(x_{new}, x_{near})) \wedge \sum_G Cost(x_{new}) + Cost(x_{new}, x_{near}) < \sum_G Cost(x_{near})$ **then** ▷ Rewiring of the tree
          $x_{parent} \leftarrow Parent_G(x_{near})$
          Replace (x_{parent}, x_{near}) in E with (x_{new}, x_{near})

In general, RRT* functions by sampling random states x_{rand} and inspecting the nearest nodes of the current tree G to find the closest node $x_{nearest}$. A new node x_{new} is created existing on a line between the sampled node and the nearest node. If the discrete path between x_{new} and $x_{nearest}$ passes the logical constraints defined in $Check\_constraint$, then the algorithm proceeds by gathering all other nodes in G within a defined radius of x_{new}, called X_{near}. The edge to x_{new} is determined by calculating the nearest tree node in X_{near} that results in the lowest cumulative cost through the tree to x_{new}. This edge and node x_{new} are added to the tree, after which the other nodes in X_{near} are rewired by examining if a connection as a child to x_{new} results in a lower cumulative sum in the tree

than their current cumulative sum.

After n number of iterations, a state tree will exist in which every leaf will the end of a branch composed of the optimal path (according to the cumulative $Cost$ function between each node, i.e. $\sum_G Cost(leaf)$) over the preceding nodes, while satisfying the $Check\_constraint$ function. As n approaches infinity, each branch ending in a leaf will be the globally optimal path from the root of the tree to the end node. Therefore, finding a path from the tree root (i.e. the initial condition) to the desired final condition is as simple as choosing the branch that ends within a radius of the final condition.

Of course, the optimal result is only possible in the infinite execution case. Fortunately, large amounts of samples within a small amount of time can provide satisfactory solutions for problems with limited constraint functions. In the cases where problems constitute complex constraint functions (e.g. cluttered environments), optimal solutions can take much longer. Furthermore, early cutoffs used to find solutions in a feasible time frame will often produce state paths that are comparatively non-smooth and difficult to implement in path planning that must satisfy kinematic/dynamic constraints (if such constraints were not included in the RRT* formulation).

To address the last issue mentioned, formulations of the RRT* algorithm with kinematic/dynamic constraints exist [33], [34], and can provide better solutions for implementing actual controls over. As stated, though, these typically require longer run times to provide solutions that approach the desired optimality conditions. For the simple static obstacle avoidance examples in [34] and [33], computation times to produce the optimal path took on the order of 10 seconds. Achieving these results often include smoothing procedures over generated paths (typically composed as optimization problems), restrict-

ing the types of constraints that can be explored. Overall, RRT* is a robust algorithm that can provide optimal results to a broad range of problems and constraints, but scaling and the introduction of greater constraints generally hurt the computation speeds of finding feasible solutions.

2.2 Machine Learning and Neural Networks

Machine learning encompasses a broad range of models and algorithms that utilize inference of patterns to perform desired tasks, usually accomplished through the use of training data. Classical machine learning is typically broken into two main schools, supervised and unsupervised training. Supervised learning utilizes labeled data, where correct input/output relationships are known. The most common examples of supervised learning include classification (e.g. labeling objects through images) and regression (e.g. prediction of values through learning input/output relationships). Unsupervised learning must infer patterns from data without knowledge of the correct output forms. Common examples include clustering algorithms and autoencoders. Both schools present major underpinnings of current trends in machine learning approaches, especially with respect to deep learning [35].

Perhaps one of the most recognizable tools in machine learning are artificial neural networks (NNs) used in deep learning. Inspired by biological neural networks, NNs are composed of numerous simple components assembled to form a complex system relating inputs to outputs. The most common NNs, called feedforward multilayer perceptron (MLP) networks, are composed of ”layers” in which the input vector to a layer (size N)

is multiplied by a matrix (size $L \times N$), added to a bias (size L), from which each element of the vector is then passed through a nonlinear activation function producing an output vector (size L). Sequential layers process the previous layer output vector as its own input vector. Therefore, the input and output of an entire NN is composed of the input to the first layer and output to the last layer. This structure is a robust setup for modeling arbitrarily complex systems.

NNs lend themselves well to supervised training problems, primarily through the development of backpropagation as applied to NNs in the 70s and 80s [36]. Backpropagation is a method of calculating the derivatives of a network's outputs with respect to its inputs. In simpler terms, this provides a method for incrementally changing the network weights and biases towards values that provide the desired correct output per training input (i.e. matching known correct input/output samples). More recent advances in parallized computing have enabled this type of training on far larger network models, enabling the use of deep neural networks (consisting of multiple, large layers) on larger, more complex problems.

NNs provide an attractive route for complex path planning purposes due to two primary reasons. First, NNs scale well to large input sets (such as direct camera feeds) and can generalize solutions well over such inputs [13], [35]. Second, output computations per input are limited primarily to the speed at which matrix computations and nonlinear functions can be applied. In practical terms, the computation costs associated with obtaining NN outputs are typically negligible, and can often be easily sped up further through parallized computing For path planning, recurrent NNs (RNNs) provide a means of fitting time-dependent behaviors through the use of memory.

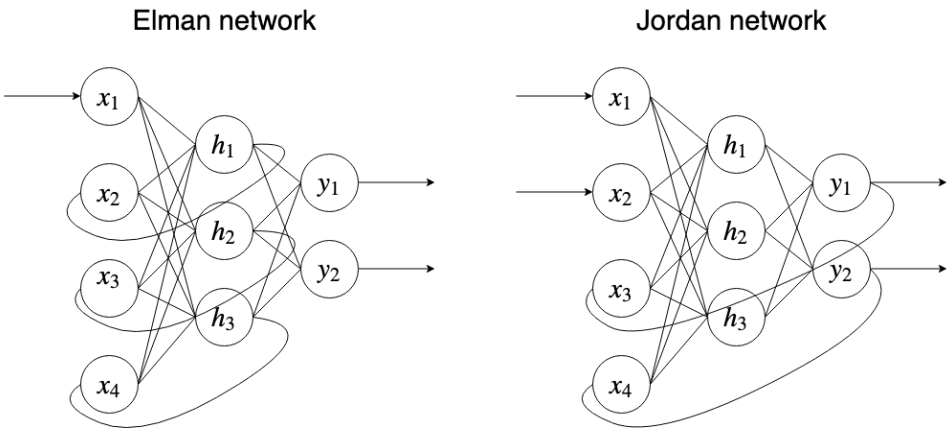

Figure 2.1: A visual comparison between the Elman and Jordan recurrent networks showcases that recurrent states of the Elman network originate from the hidden network layers while the recurrent states of the Jordan network are directly fed from the output to the input.

2.2.1 Recurrent Neural Networks

Recurrent neural networks (RNNs) are a class of NNs in which input components to any number of layers are comprised of outputs from a subsequent layer. RNNs provide the added benefit of memory to input/output relationships, i.e. previous input/output relationships impact subsequent relationships. This implies that RNNs are well suited to learning input/output relationships that are dependent on time, a primary component to path planning. Two of the simplest RNN forms are the Elman network and the Jordan network (visually represented in Fig. 2.1). Mathematically, the Elman network is represented by Eq. (2.1) and the Jordan network is represented by Eq. (2.2),

$$\begin{aligned} \mathbf{h}(t) &= \sigma_h(W_{s,h}\mathbf{x}_t + W_{i,h}\mathbf{h}_{t-1} + \mathbf{b}_h) \\ \mathbf{y}(t) &= \sigma_y(W_{s,y}\mathbf{h}_t + \mathbf{b}_y), \end{aligned} \tag{2.1}$$

$$\begin{aligned} \mathbf{h}(t) &= \sigma_h(W_{s,h}\mathbf{x}_t + W_{i,h}\mathbf{y}_{t-1} + \mathbf{b}_h) \\ \mathbf{y}(t) &= \sigma_y(W_{s,y}\mathbf{h}_t + \mathbf{b}_y), \end{aligned} \tag{2.2}$$

where $\mathbf{x}_t$ is the input, $\mathbf{h}_t$ is the hidden layer output, $\mathbf{y}_t$ is the network output, W_s, W_i, and $\mathbf{b}$ are parameter matrices and vectors, and t refers to the time step.

Both networks serve as the basis for larger and more complex RNN structures, including Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) networks. This thesis focuses on the use of networks extended from the Jordan network form, specifically the Nonlinear autoregressive exogenous (NARX) network, of the form described in Eq. (2.3),

$$\begin{aligned} \mathbf{x}_t &= \Phi(\mathbf{x}_{t-1}, \mathbf{u}_t), \\ \text{where } \Phi(\cdot) &= \sigma_1 \circ \sigma_2 \circ ... \sigma_L(\cdot), \\ \text{with the form } \sigma_l(\cdot) &= \sigma_l(W_l\mathbf{h}_l + \mathbf{b}_l). \end{aligned} \tag{2.3}$$

NARX networks lend themselves well to time-dependent state training sets due to their explicit recurrent layer, i.e. only network outputs of the last layer directly feed back into the network inputs alongside environment inputs. As a result, training can easily utilize many training schemes without the use of backpropagation through time (BPTT) [37], necessary for training RNNs with internal recurrent layers. BPTT introduces added complexities due to gradients dependent upon previous layer outputs, amplifying the issue of error gradients vanishing (which reduces the ability to train the network). Additionally, internal recurrent layers add further complexities to the use of bounded set propagation for RNNs, another topic of this thesis. Due to these two points, NARX networks maintain the easiest approach for this thesis's purposes.

2.2.2 Supervised Network Training

Multiple training schemes exist to train sets of NN inputs, X_{in}, over desired output sets, Y_{out}. At the heart of these algorithms is the idea of optimizing the NN with respect to a loss function over the produced output $Y_{nn} = \Phi(X_{in})$ and truth output Y_{out}. A common loss function for optimizing the NN with respect to is the Euclidean distance, i.e. $E = \frac{1}{2} \|\Phi(X_{in,i}) - Y_{out,i})\|^2$, where the index i refers to individual data points in each set. In most training schemes, derivatives of the NN with respect to its weights and biases are calculated with respect to these loss functions (e.g. $\frac{\partial E}{\partial w}$, $\frac{\partial E}{\partial b}$) and used to incrementally modify the NN weights and biases as to move towards a setup that minimizes the loss function across all inputs. The exact methods used in these approaches are beyond the scope of this thesis, and numerous sources exist to provide detailed overviews [38]. Common training schemes include Stochastic Gradient Descent (SGD) [39] and ADAM [40], each offering varying performances for any given problem.

With respect to NARX RNNs fitted to time-dependent series, training data is comprised of sequential input, and output pairs. This means that a sample input/output pair containing the variables x_i/x_{i+1} is followed by another sample input/output pair containing x_{i+1}/x_{i+2}. In regards to training, the previously discussed types of training schemes offer adequate performance *with respect to the provided data points*. This means that closed-loop performance of an RNN (vital to a path planning RNN) producing the same time-dependent sequence of states as the training data (initialized with an input containing x_0 to produce a predicted state x_1 which is fed back into the network, recursively) is highly dependent on the prediction quality of each individual data point. What results is

that accumulated errors can quickly and easily result in a generated sequence of states that diverge from the desired path. In general, supervised training alone is not enough to produce a NARX RNN that can reasonable predict desired time-dependent series, and other ideas in training NNs must be introduced.

2.2.3 Reinforcement Learning

For inspiration to improving the ability of a NARX RNN to predict a desired state sequence, this thesis examines concepts behind reinforcement learning (RL). In general RL [41], an agent (such as an NN) is formulated as a policy that maps an environment of perception inputs to action outputs. From such, given an environment, an agent is allowed to produce an action and is trained to improve the action's result in the environment through the use of a reward function. RL with deep neural networks has found large successes in recent years, resulting in the ability of such constructs to play video games [42], drive simulated cars [43], control nonlinear systems [44] and more. RL without an explicit model differs inherently from supervised learning in that no correct input/output pairs are known and trained against. Instead training is performed to drive the NN to produce actions that maximize the reward function over an infinite time horizon (composed of sequences of action outputs produced by the agent, resulting state changes due to the action, and followed by new perception inputs for the next time step). While deep RL proposes an exciting and developing research topic, it also poses a difficult model for study since the more common model-less RL treats the trained NN as a black box, which presents greater difficulties in verification.

2.3 Formal Verification and Bounded Set Propagation

Formal verification is the approach of automatically verifying desired properties of systems (involving hardware and/or software) through the use of formal methods. Formal methods consist of numerous mathematical and computer science techniques for defining properties and systems, proving or disproving properties of systems, and synthesizing systems from properties. [45] provides a fairly large scale overview of this field.

Formal verification of NN properties is a rapidly growing field, primarily due to the fact that supervised training of NNs is built on the concept of using verified results to create a system that can generalize to new input/output pairs. This notion is at contrast with the complexity of NNs and the difficulty in accounting for all possible input/output relationships. As previously mentioned in Chapter 1.1, numerous sources have made strides towards methods of verifying numerous types of NNs for various problems. With respect to NARX RNNs, property verification is typically concerned with examining the sets of sequences of recurrent states under environment inputs. As a result, the concept of bounded set propagation lends itself well to calculating all possible recurrent states for a given set of inputs.

Bounded set propagation is composed of the following problem. Provided an input boundary set $H = \{\eta \in \mathbb{R}^n | \underline{\eta} \leq \eta \leq \overline{\eta}\}$ and input/output system $\zeta = \Phi(\eta)$ (e.g. a NN), what is the resulting output bounded set $Z = \{\zeta \in \mathbb{R}^m | \underline{\zeta} \leq \zeta \leq \overline{\zeta}\}$? Bounded set propagation is an important tool in reachability analysis and collision detection of systems, as observed especially for dynamical systems under external noises [46]. This tool is an important step in property verification of systems that contain complex input/output re-

lationships, and provides an important piece of future formal method suites for verifying NNs.

Chapter 3: Methodology and Problem Description

Chapter 3 focuses on the methodology presented in this thesis and the formal problem descriptions for each step.

3.1 Methodology Overview

The methodology presented in this paper is built around a framework for generating RNN path planners for system states from initial configurations to end configurations under a broad range of environment and kinematic/dynamic constraints and optimization metrics. The state path generated by the RNN is to be used in a path tracking controller, of which the resulting control signals trend that of the desired optimal control signal. This methodology is visualized in Fig. 3.1.

The driving goal of this methodology is to create a versatile, computationally fast path planner for generically defined kinematic/dynamic systems that is capable of providing feasible solutions to varied environments while still maintaining optimization of the desired metric. The greatest strength in this approach is that the use of RNNs in learning the solution space of an optimization problem results in order of magnitude reductions in the computation costs of generating paths. For comparison, the generation of feasible paths from a kinodynamic RRT* may require up to 10 seconds, while sequential RNN

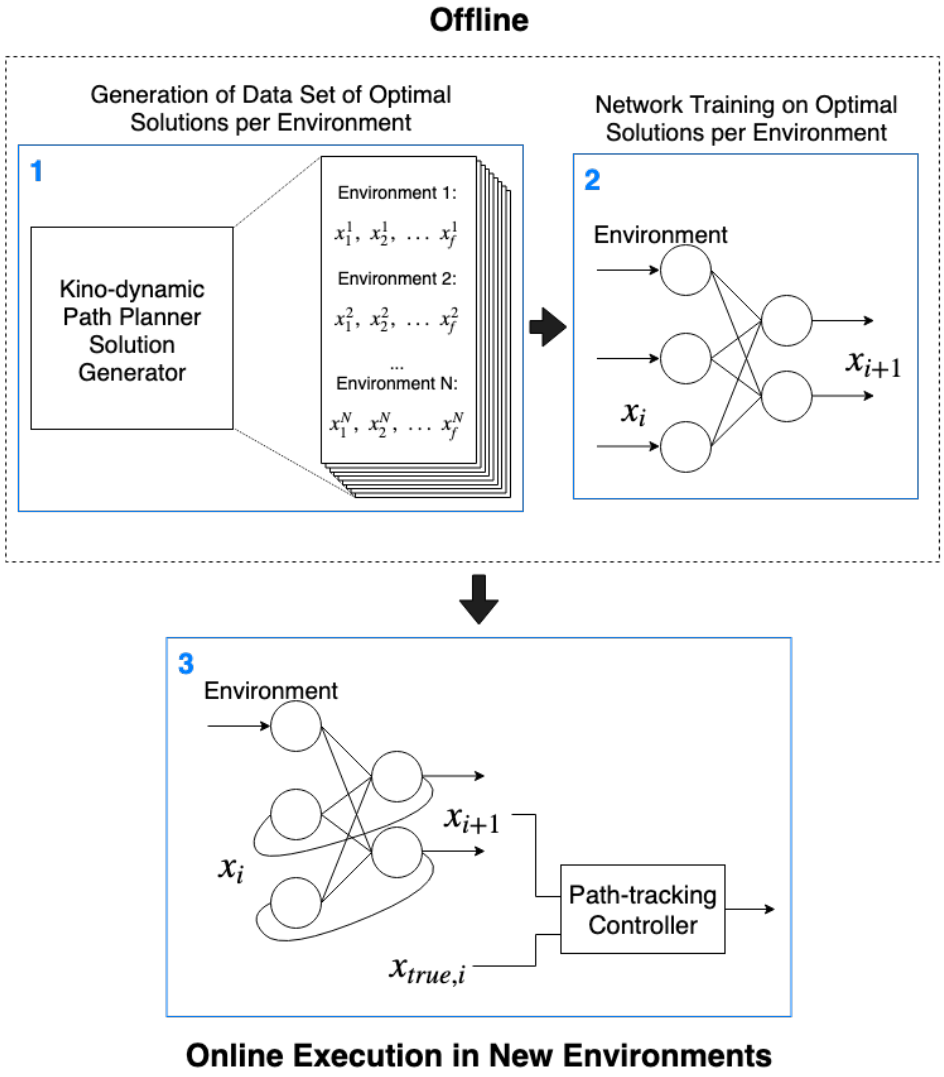

Figure 3.1: Overview diagram of the three main components of the methodology explored. For part 1, a kinodynamic path planner solution generator is used offline to generate large quantities of optimized solutions under varied environments. From such data, a recurrent neural network (RNN) is trained in part 2 to recreate the state paths of the optimized solutions, with the ability to generalize the solutions to new environments. For the online portion of part 3, the RNN is utilized to generate the state path of the optimal solution, from which a path tracking controller is used to follow. Control signals of the path tracking controller will trend that of the optimized solution.

computations will take on the order of tens of milliseconds. The primary hurdle, though, is accurately teaching the RNN to consistently produce feasible results.

The following sections outline the formal problem formulations of each step in the methodology.

3.2 Problem Formulations

The three components of the methodology, Path Planning, RNN Training, and Controller Execution, are formulated as individual problems to be discussed and solved in Chapter 4 of this thesis.

3.2.1 Path Planning under Kinematic/Dynamic and Environment Constraints

The generalized path planning problem is formulated as the optimal control problem presented in the following way:

$$\underset{\mathbf{u}(t),t_f}{\text{minimize}} \quad \int_0^{t_f} \mathscr{W}(t,\mathbf{x}(t),\mathbf{u}(t))dt + \mathscr{L}(\mathbf{x}(0),t_f,\mathbf{x}(t_f)), \tag{3.1a}$$

$$\text{subject to} \quad \dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}(t),\mathbf{u}(t)), \tag{3.1b}$$

$$\mathbf{x}(0) = \mathbf{x}_0, \tag{3.1c}$$

$$\mathbf{x}(t_f) = \mathbf{x}_f, \tag{3.1d}$$

$$\mathbf{C}(\mathbf{x}(t),\mathbf{u}(t),\mathbf{P}) \leq \mathbf{0}. \tag{3.1e}$$

Here, $\mathbf{x}(t) \in \mathbb{R}^N$ is the system state, $\mathbf{u}(t) \in \mathbb{R}^M$ is the control signal, and t is time. Eq.

(3.1a) is an optimization metric (consisting of both an integrated scalar function $\mathscr{W}$ and non-integrated scalar function $\mathscr{L}$) for the provided kinematic/dynamic system defined by Eq. (3.1b), in which $\mathbf{f}(\mathbf{x}(t),\mathbf{u}(t)) \in \mathbb{R}^N$. Eq. (3.1c) and (3.1d) represent desired initial and final conditions on the state, respectively, and Eq. (3.1e) (with $\mathbf{C}(\mathbf{x}(t),\mathbf{u}(t),\mathbf{P}) \in \mathbb{R}^Q$) contains all desired nonlinear constraints on the system states and control signals. The vector $\mathbf{P} \in \mathbb{R}^O$ represents all possible static variables in the constraints. A control solution to the above optimization formulation and its corresponding state path is represented as $\mathbf{G}(t) \in \mathbb{R}^{N+M}$.

3.2.2 Recurrent Neural Network and Training

Provided a domain for the initial and final conditions, $\mathbf{x}_{0,min} \leq \mathbf{x}_0 \leq \mathbf{x}_{0,max}$ and $\mathbf{x}_{f,min} \leq \mathbf{x}_f \leq \mathbf{x}_{f,max}$, and a constraint domain of $\mathbf{P}_{min} \leq \mathbf{P} \leq \mathbf{P}_{max}$, individual optimized solutions $\mathbf{G}(t)$ exist as outputs to the optimization solution when using these variables. Provided sets X_0, X_f, and P_{set} of sample points from these domains, a set of solutions G_{set} exists, composed of the solutions to the optimization problem described in the previous section using these variables.

Given G_{set}, an RNN must be formed and trained upon the provided data, operating under fixed time-step t_k. The RNN is represented generally as the form,

$$\mathbf{x}(t_k+1) = \Phi(\mathbf{x}(t_k), t_k, \mathbf{x}_i, \mathbf{x}_f, \mathbf{P}), \tag{3.2}$$

where t_k represents sampled time. For all given time steps and for all solutions $\mathbf{G}(t) \in G_{set}$, the RNN output must be trained to minimize a loss function (typically the Euclidean

distance squared) composed of the term,

$$\Phi(\mathbf{G}_x(t_k), t_k, \mathbf{x}_i, \mathbf{x}_f, \mathbf{P}) - \mathbf{G}_x(t_k + 1), \tag{3.3}$$

where $\mathbf{G}_x(t)$ is the state component of a given solution vector $\mathbf{G}(t)$.

3.2.3 Executed Controller

Provided a state path $\sigma(t_k) : \mathbb{T} \longrightarrow \mathbb{R}^N$ generated by closed-loop execution of the RNN under set values of $\mathbf{x}_i$, $\mathbf{x}_f$, and $\mathbf{P}$ with $\|\mathbf{x}_i - \sigma(0)\| \leq \delta$, where δ is an arbitrarily small number, a controller $\mathbf{u}_e(t, \mathbf{x}(t), \sigma(t_k))$ must be formulated such that the error norm $\|\mathbf{x}(t) - \sigma(t_k)\|$ is minimized while all constraints $\mathbf{C}(\mathbf{x}(t), \mathbf{u}(t), \mathbf{P}) \leq \mathbf{0}$ are satisfied. Additionally, the control signal $\mathbf{u}_e$ should mimic that of the true control signal $\mathbf{G}_u(t)$, minimizing the error between the optimized control signal and the executed control signal.

Chapter 4: Methodology Components

Chapter 4 outlines the solutions utilized to solve the problem formulations presented in Chapter 3.2.1 and 3.2.2, the path planning and network training problems.

4.1 Kinodynamic Optimal Rapidly-exploring Random Tree with Chebyshev Polynomial Collocation Optimization

Originally explored in [33] for linear systems and extended to differentially flat systems in [34], kinodynamic formulations of RRT* follow a similar format as the original RRT*. The primary differences between the two formulations typically include the *Cost* function computation (a function of the state, control, and time for kinodynamic RRT*) and the method for computing optimal state and control solutions between tree nodes and random samples (solving optimal control problems with respect to the cost function). Algorithm 2 provides the general kinodynamic RRT* formulation, where the contribution of this thesis is the extension of solving the nonlinear problem formulations between nodes by using Chebyshev collocation approaches.

In Algorithm 2, $Cost()$ represents Eq.(3.1a) evaluated over a branch and $BVP()$ is the solution to the boundary value problem solved as an optimization problem between two states using Eq. (3.1a)-(3.1d). Notice that solutions solved between two individual

states do not require nonlinear constraint formulations of Eq. (3.1e). The primary difficulty in implementing the kinodynamic RRT* formulation is the method in which $BVP()$ is calculated between two state nodes.

Algorithm 2: Kinodynamic RRT*

procedure GENERATE_PATH(x_{init}, N) ▷ Execute RRT* over N iterations
  $V \leftarrow \{x_{init}\}$; $E \leftarrow \emptyset$
  $x_{prev} \leftarrow x_{init}$
  for $i = 1, \ldots, n$ **do**
    Attempt $e \leftarrow BVP(x_{prev}, x_f)$
    if $Cost(e) < cost_{connect}$ **then**
      $V \leftarrow V \cup x_f$; $E \leftarrow E \cup e$
    $x_{rand} \leftarrow$ Sample state from regions that don't violate constraints
    $x_{nearest} \leftarrow \text{argmin}_{v \in V} Cost(BVP(v, x_{rand}))$
    $x_{new} \leftarrow x_{rand}$
    if $Check\_constraint(BVP(x_{nearest}, x_{new}))$ **then**
      $X_{near} \leftarrow \{v \in V \mid Cost(BVP(v, x_{new})) \leq cost_{radius}\}$
      $V \leftarrow V \cup \{x_{new}\}$
      $x_{min} \leftarrow x_{nearest}$; $c_{min} \leftarrow \sum_G Cost(x_{nearest}) + Cost(BVP(x_{nearest}, x_{new}))$
      for $x_{near} \in X_{near}$ **do**
        if $Check\_constraint(BVP(x_{near}, x_{new})) \wedge \sum_G Cost(x_{near}) + Cost(BVP(x_{near}, x_{new})) < c_{min}$ **then** ▷ Determine minimum cost connection
          $x_{min} \leftarrow x_{near}$; $c_{min} \leftarrow \sum_G Cost(x_{near}) + Cost(BVP(x_{near}, x_{new}))$
        $E \leftarrow E \cup \{BVP(x_{min}, x_{new})\}$
      for $x_{near} \in X_{near}$ **do**
        if $Check\_constraint(BVP(x_{new}, x_{near})) \wedge \sum_G Cost(x_{new}) + Cost(BVP(x_{new}, x_{near})) < \sum_G Cost(x_{near})$ **then** ▷ Rewiring of the tree
          $x_{parent} \leftarrow Parent_G(x_{near})$
          Replace $BVP(x_{parent}, x_{near})$ in E with $BVP(x_{new}, x_{near})$
      $x_{prev} \leftarrow x_{new}$

The methods used in $BVP()$ have varied between the kinodynamic RRT* formulations. For linear systems, [33] exploited classical control theory to determine the weighted controllability Gramian of the linear system used in computing the optimal control policy from an initial state to a final state over fixed time. From here, this optimal control policy substituted into the cost function allowed for an analytical derivation of the cost function's

derivative with respect to time. By using this derivative to minimize the cost with respect to time, [33] found an analytical form of the optimal control history that minimized the cost function with respect to time. The primary limitations of this method is the required linearity of the defined system and a cost function solely defined as $\int_0^{t_f}(1+\mathbf{u}(t)^T\mathbf{u}(t)dt)$ (where $\mathbf{u}$ is the control signal and t_f is final time).

For differentially flat systems (in which a flat output $y = h(x,u,\dot{u},...u^{(k)})$ exists for system $\dot{x} = f(x,u)$ resulting in the existence of functions $x = g(y,\dot{y},...,y^{(j)})$ and $u = g'(y,\dot{y},...,y^{(j)})$), [34] utilized B-splines [47] to represent state trajectories between tree nodes and sampled points. The use of B-splines allowed the optimal control problem to be formulated as a nonlinear programming (NLP) problem, from which nonlinear optimizers could be used to solve for. Optimized state trajectories utilizing B-splines require differentially flat systems, resulting in non-explicit formulations of the states separated from the controls. This results in a state and control representation that are tied directly together (through the supposed function g), not easily allowing for optimal solutions that may contain controls spikes separate from the optimal state path. For example, bang-bang control solutions are present in low-thrust spacecraft trajectory optimization, but optimization through assumptions of the system as differentially flat do not easily allow for representations that produce a discontinuous control signal alongside explicit control constraints. Circumventing the issue of assuming differentially flat system definitions required the use of analytic homotopic approaches as auxillary control solutions in [48], for example.

This thesis explores the use of Chebyshev polynomial representations in solving the optimization problem for $BVP()$ through collocation (also referred as psuedospectral

methods) as explored in [49] and [50], which do not require explicitly defined differentially flat systems. In this approach, Chebyshev polynomials of the form

$$C_N = \cos\left(N_t \cos^{-1}(\tau)\right), \tag{4.1}$$

where $N_t + 1$ represents the number of collocation nodes and $\tau \in \{-1, 1\}$, serve as representations of the state over the optimization horizon. At discrete $N_t + 1$ nodes of the polynomial, formulated from the chosen values of τ as

$$\tau_k = -\cos\left(\frac{\pi k}{N_t}\right) \quad k = \{0, 1, ..., N_t\}, \tag{4.2}$$

the polynomial's time derivative is constrained to equal that of the state's dynamics. Under this discretization of the state path as a Chebyshev polynomial, the optimization problem for $BVP()$ is reformulated as the following:

$$\underset{U_d, X_d, t_{N_t}}{\text{minimize}} \quad \sum_{k=0}^{N_t} \left(w_k \mathscr{W}(\tau_k, \mathbf{x}(\tau_k), \mathbf{u}(\tau_k))\right) + \mathscr{L}(\mathbf{x}(0), t_{N_t}, \mathbf{x}(\tau_{N_t})), \tag{4.3a}$$

$$\text{subject to} \quad DX_d - F = 0, \tag{4.3b}$$

$$\mathbf{x}(0) = \mathbf{x}_0, \tag{4.3c}$$

$$\mathbf{x}(\tau_{N_t}) = \mathbf{x}_f \tag{4.3d}$$

In the above equations, $N_t + 1$ is the number of discrete nodes. $X_d \in \mathbb{R}^{(N_t+1)\times N}$, $U_d \in \mathbb{R}^{(N_t+1)\times M}$, and $F \in \mathbb{R}^{(N_t+1)\times N}$ are matrix representations of the discrete nodes of the Chebyshev polynomial and the corresponding state derivative and constraint evaluations

at each, explicitly written as:

$$X_d = \begin{bmatrix} x_1(0) & x_2(0) & \dots & x_N(0) \\ x_1(\tau_1) & x_2(\tau_1) & \dots & x_N(\tau_1) \\ \vdots & \vdots & \ddots & \vdots \\ x_1(\tau_{N_t}) & x_2(\tau_{N_t}) & \dots & x_N(\tau_{N_t}) \end{bmatrix}, \tag{4.4a}$$

$$U_d = \begin{bmatrix} u_1(0) & u_2(0) & \dots & u_M(0) \\ u_1(\tau_1) & u_2(\tau_1) & \dots & u_M(\tau_1) \\ \vdots & \vdots & \ddots & \vdots \\ u_1(\tau_{N_t}) & u_2(\tau_{N_t}) & \dots & u_M(\tau_{N_t}) \end{bmatrix}, \tag{4.4b}$$

$$F = \frac{t_{N_t}}{2} \begin{bmatrix} f_1(\mathbf{x}(0),\mathbf{u}(0)) & \dots & f_N(\mathbf{x}(0),\mathbf{u}(0)) \\ f_1(\mathbf{x}(\tau_1),\mathbf{u}(\tau_1)) & \dots & f_N(\mathbf{x}(\tau_1),\mathbf{u}(\tau_1)) \\ \vdots & \ddots & \vdots \\ f_1(\mathbf{x}(\tau_{N_t}),\mathbf{u}(\tau_{N_t})) & \dots & f_N(\mathbf{x}(\tau_{N_t}),\mathbf{u}(\tau_{N_t})) \end{bmatrix}, \tag{4.4c}$$

where the scaling term $t_{N_t}/2$ in the state derivative is introduced due to state's transformation onto the time domain expressed in Eq. (4.2). The matrix $D \in \mathbb{R}^{(N_t+1)\times(N_t+1)}$ is the differentiation matrix of the Chebyshev polynomial, explicitly written as,

$$D_{i,k} = \begin{cases} \frac{a_k(-1)^{k+i}}{a_i(\tau_k-\tau_i)} & \text{if } k \neq i \\ -\frac{\tau_k}{2(1-\tau_k^2)} & \text{if } 1 \leq k = i \leq N_t - 1 \\ \frac{2N_t^2+1}{6} & \text{if } k = i = 0 \\ -\frac{2N_t^2+1}{6} & \text{if } k = i = N_t - 1, \end{cases} \tag{4.5}$$

where $a_{k,i} = 2$ if $k,i = \{0,N_t\}$ and $a_{k,i} = 1$ otherwise. The quadrature weights w_k are used in approximating the integral of a function evaluation on a Chebyshev polynomial,

formulated as,

$$w_k = \frac{c_k}{N_t}\left(1 - \sum_{j=1}^{N_t/2} \frac{b_j}{4j^2-1}\cos(2j\tau_k)\right), \tag{4.6}$$

where $b_j = 2$ if $j = N_t$ and $b_j = 1$ otherwise. The variable $c_k = 1$ if $k = \{0, N_t\}$ and $c_k = 2$ otherwise. Under the presented formulation, the discrete values in U_d, X_d and t_{N_t} make up the free parameters for an NLP program, alongside the provided constraints and optimization function.

This formulation described does not require differentially flat systems and allows for control solutions separated from state paths, where these control solutions may closer represent possible discontinuities. Common NLP solvers, such as ones utilizing sequential quadratic programming (SQP), are typically well suited for solving these types of formulations. Fig. 4.1 provides an example of the kinodynamic RRT* finding a solution for a path planning problem (utilizing $F = ma$ dynamics) in a cluttered environment.

The use of the Chebyshev collocation optimization in the kinodynamic RRT* allows for the generation of large amounts of optimized solutions for a given problem scenario. The generation of these data sets leads into the development of a method for training the path planning RNN on the optimized solutions, discussed in the following section.

4.2 Whole-path Reinforcement Training Scheme

The second component of the methodology presented in this thesis requires training the RNNs to learn the state solution space of the solutions generated by the kinodynamic RRT*. Supervised training of RNNs to predict time-dependent state sequences produces

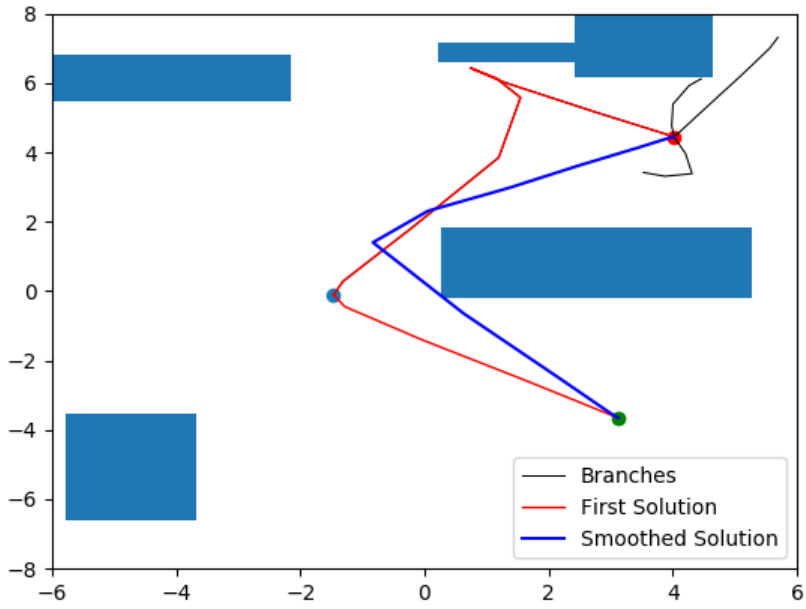

Figure 4.1: The results of a kinodynamic RRT* run after 5 iterations. The path planning problem consists of obstacle avoidance in a cluttered environment for a 2D point mass dictated by $F = ma$ dynamics, starting at the red dot and ending at the green dot. The black branches represent sampled paths of the RRT*, while the red branch is the first feasible solution found. This solution was used as an initial guess to an NLP solver to create a smoothed solution, represented by the blue line.

networks that drift from the desired output over time when in closed-loop execution. This is primarily due to the accumulation of errors in the state outputs over time. For example, suppose a double pendulum is simulated from an initial condition to create a sequence of state paths (including joint orientations and velocities) over a fixed time. A simple RNN is created as the form $x_{i+1} = \Phi(x_i)$, where Φ represents a simple feedforward network. In a supervised training scheme, the training data set would be constructed as $\{(x_i, x_{i+1})\}$ for $i = \{1, 2, ...N-1\}$, where the states x_i are sampled at discrete points in time. Training the RNN under a typical supervised fashion (such as with the use of SGD), results in Fig. 4.2. Initial errors in the state prediction quickly lead to a breakdown in the ability to provide a reasonable prediction of the state paths over the execution.

The provided example illustrates that an improved training scheme is required for producing RNNs that can reliably predict state paths of complex systems. To achieve this goal, RL concepts can help provide guidance. In RL, outputs of NNs are fed into systems which impact the inputs for the NN. This feedback loop can be exploited to train the network based on the results its actions have on the environment. In many RL applications, the training is approached through the maximization of a reward function, where schemes are set up to train NN weights in the direction that produces actions which increase the reward. For RNN predictions of time-dependent paths, the closed-loop execution of the RNN can be exploited to force the resulting outputs back towards the desired path. This insight led to the contribution of Algorithm 3.

Algorithm 3 takes in a set G_{set} of optimized state path solutions of Eq. (3.1a) - (3.1b) and constructs the time-dependent training data (X_{in}, Y_{out}). A normal supervised training procedure is first employed to prime the RNN in fitting the desired paths. Through mul-

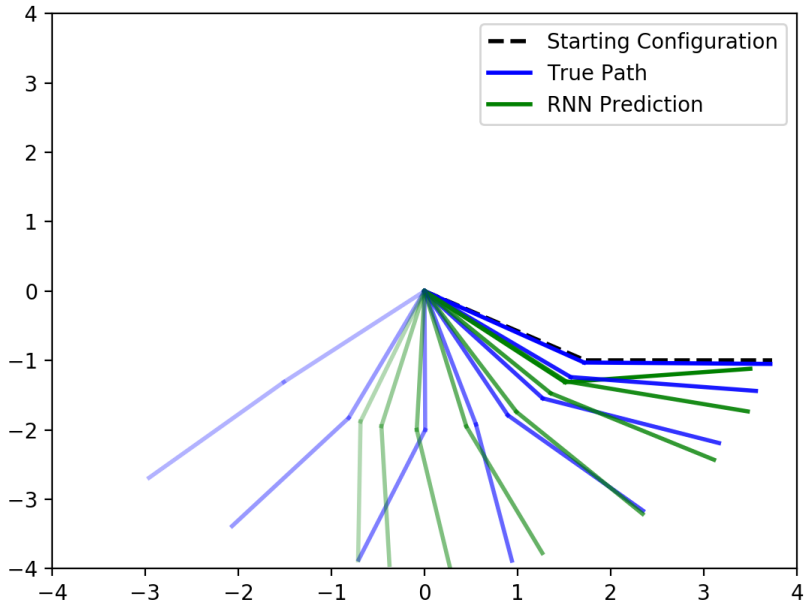

Figure 4.2: Under standard training schemes, errors in the predicted states of an RNN accumulate over time, leading to worse path predictions. The figure showcases the starting configuration and path of a double pendulum alongside the predicted path from the RNN. Transparency of the pendulum increases forward in time. Pendulum masses of 4 kg and lengths of 2 m were used in modeling the system.

Algorithm 3: Whole-path Reinforcement Training

procedure TRAIN_RNN(G_{set}) ▷ Train RNN from set of optimized solutions
    Train RNN over $(X_{in}, Y_{out}) \subseteq (G_{set}(t_k), G_{set}(t_k+1))$ for all k ▷ Utilizing any common training scheme
    $(X_{in}, Y_{out})_{original} \leftarrow (X_{in}, Y_{out})$
    $MSE_{best} \leftarrow$ inf
    for 1 to $Training\_Iterations$ **do**
        for $i \leftarrow 1 : len_{horizon} : N_{t_f}$ **do**
            Generate $\sigma(t_k)$ from $k = 0$ to $k = i$ for all x_i, x_f, and P sets
            Create training data $(X_{horizon}, Y_{horizon}) = (\sigma(t_k), G_{set}(t_k+1))$ from $k = 0$ to $k = i$
            $(X_{in}, Y_{out}) \leftarrow (X_{horizon}, Y_{horizon}) \cup (X_{in}, Y_{out})$
            Train RNN over (X_{in}, Y_{out})
        Generate $\sigma(t_k)$ from $k = 0$ to $k = N_{t_f}$ for all x_i, x_f, and P sets
        $MSE_r \leftarrow$ mean square error of RNN output w.r.t $G_{set}(t_k+1)$ given $\sigma(t_k)$ input set
        if $MSE_r < MSE_{best}$ **then**
            $MSE_{best} \leftarrow MSE_r$
            $RNN_{best} \leftarrow RNN$

tiple training iterations, closed-loop execution of the RNN is used to produce the set of state paths $\sigma(t_k)$ over finite time horizon. An augmented training set $(X_{horizon}, Y_{horizon})$ is constructed of the state paths $\sigma(t_k)$ produced by the RNN and the desired path set $G_{set}(t_k+1)$ at the next time step for all t_k. This set is appended to the total training set, and the RNN is trained over the total training set again. After the longest time horizon is reached, the quality of the RNN at the current iteration is assessed through comparison of its closed-loop execution with respect to the desired path outputs. This is performed through the loss function comparison, which is assumed to be the mean square error of the path predictions against the desired path outputs. If the loss calculation is less than the best value, the RNN is saved as RNN_{best} alongside the current mean square error value. From here, the process repeats for the desired number of $Training\_Iterations$.

The results of the proposed algorithm are improved path predictions across the data

set provided. To be clear, though, this form of learning does not produce RNNs that learn underlying dynamics, since individual states at each time step are corrected to follow a desired path output. The RNN instead learns a path structure from the data set. For the purposes of the methodology discussed in this paper, such an outcome is desired. The RNN is supposed to learn optimal solutions *per environment input*. Because only one optimal solution exists per environment, strict enforcement of path learning for each optimized solution helps to extend the RNN's capabilities in generalizing across new permutations in the environment, instead of across different states for individual environments. The improvements that Algorithm 3 can produce are shown in the application to the pendulum problem previously discussed, shown in Fig. 4.3. This level of improvement helps improve the RNNs path planning capabilities, a necessary component in the overall methodology.

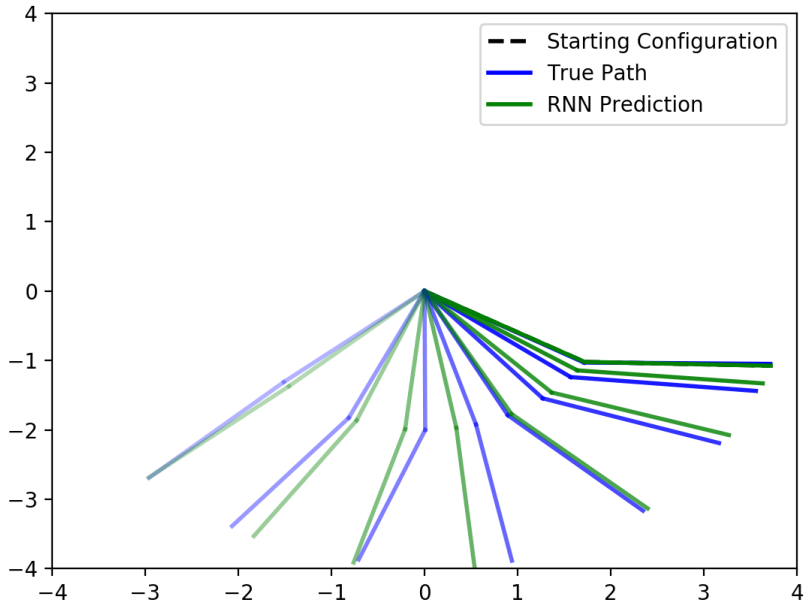

Figure 4.3: Under the whole-path reinforcement training scheme of Algorithm 3, errors in the predicted states of an RNN do not accumulate nearly as much over time. In many cases, recent errors may not impact future predictions since the RNN learned the path structure through the entire time duration of this scenario. The figure showcases the starting configuration and path of a double pendulum alongside the predicted path from the RNN. Transparency of the pendulum increases forward in time. Pendulum masses of 4 kg and lengths of 2 m were used in modeling the system.

Chapter 5: Problem Scenario 1: 2D Obstacle Avoidance

To examine the effectiveness of the methodology proposed in this thesis with respect to highly constrained environments, a cluttered static obstacle avoidance problem with 2D dynamics is examined. The problem formulation and application of the methodology with respect to it is further described in the following sections.

5.1 Implementation

Path planning between two locations in a cluttered environment without kinematic-/dynamic considerations presents a difficult problem to solve, one further complicated by the introduction of kinematic/dynamic constraints and optimality conditions. Introduced previously in Chapter 4.1, a kinodynamic RRT* can provide suboptimal solutions quickly (order of 0.25 seconds in [34]) and provide optimal solutions under longer computation times (order of 10 seconds in [34]). The capability to provide a feasible solution relatively close to the desired optimal solution in an order of magnitude less time than even the suboptimal solution is of great interest, a primary result this thesis's methodology is aimed to create. To explore such, the problem formulation (described in respect to the optimization formulation in Chapter 3) and implementation details are provided.

5.1.1 Problem Definition and Application Details

For this 2D point-to-point problem with obstacle avoidance and dynamics, the following model utilized is,

$$\dot{\mathbf{x}} = \mathbf{v} \tag{5.1}$$

$$\dot{\mathbf{v}} = \mathbf{u}_e m, \tag{5.2}$$

where $m = 1$. The state boundary is $(-6,-8) \leq (x,y) \leq (6,8)$ and $(-2.5,-2.5) \leq (v_x,v_y) \leq (2.5,2.5)$, with control constraints of $(-10,-10) \leq (u_x,u_y) \leq (10,10)$. The environment consists of 7 rectangular obstacles randomly placed within the state domain, each formulated as,

$$(-6,-8,0.5,0.5) \leq \mathbf{P}_i = (x_p,y_p,h_p,w_p) \leq (6,8,5,5), \tag{5.3}$$

where x_p and y_p are position coordinates of the rectangle's center and h_p and w_p are heights and widths, respectively. Initial and final state positions (with zero velocity) are sampled randomly within the state domain and outside of obstacles. The optimization function to minimize for a path is,

$$\int_0^{t_f} \|\mathbf{u}_e\| \, dt + t_f. \tag{5.4}$$

With respect to the first step of the methodology, the kinodynamic RRT* utilizing the Chebyshev collocation optimization scheme was used in obtaining training data for an RNN learning the solution space to this problem. Approximately 5,000 solutions were generated for training and validation (70/30 split), consisting of 10 final positions x_f

per 10 initial positions x_i per 50 random object sets P_i. The solver SNOPT [51] was utilized for solving each branch of the kinodynamic RRT* and performing the smoothing procedure for each solution. For the cluttered environment described, this data gathering process took on the order of 3 days run-time with a 2.5 GHz processor.

The creation and training of an RNN over this data set utilized the Keras [52] and TensorFlow [53] libraries in Python. The network architecture consisted of an input layer (consisting of state and environment variables), 4 hidden layers (sizes 70/50/50/50) with hyperbolic arctan activations, and a linearly activated output layer for the state. The sampling time of the RNN was 0.1 seconds (i.e. t_k = 0.0, 0.1, 0.2,...). Network training over any data set utilized the ADAM optimization scheme with Nesterov momentum integrated [54] and a learning rate of 0.002. In addition to the standard environment representation using the vector $\mathbf{P}_i$ as input, an autoencoded representation for the obstacles was also explored to assess how varied environment representations may improve training performance.

5.1.2 Contractive Autoencoding of Environment

Contractive autoencoders [55] are feedforward neural networks that encode high density information to lower density outputs. These models are trained in an unsupervised fashion, where a feedforward network is used to encode the information to a lower density form, from which this output is fed into the inverse of the network used to encode such. Utilizing the resulting output, the encoding-decoding network combination is trained to replicate the input exactly with its output. The resulting encoding network acts

as a compression system, from which decompression is performed through the decoding network form.

With respect to the obstacle avoidance problem at hand, obstacles normally represented as 4 dimensions (location, height, width) can instead be represented by point clouds. An autoencoder acting on the high-dimension point cloud can be used to create a reduced representation of this environment for use by the RNN. This reduced environment presents a more robust representation, since separate training data sets that normally utilize the same obstacles (across different initial and final state conditions) will instead use encoded representations that differ slightly between each other. This results in a more robust training sequence since a larger set of environment representations are observed. The motion planner in [31] reported significant training improvements when utilizing this scheme for representing the environment. For the results of this Problem Scenario, assessments are made on how well the autoencoding of the environment can improve results.

5.1.3 Path-tracking Controller

Last, a path-tracking controller was formulated to fulfill the final part of the methodology: online execution. This controller was constructed after examining preliminary results in order to mitigate common errors observed in the RNN path planner. In general, design of a controller to track the generated path $\sigma(t)$ of the RNN is a problem-specific task tied to the kinematics/dynamics of the prescribed system. The ability to track an arbitrary path provided a system and control definition is dependent upon the controllability of the system and realizability of a reference track [56]. Fortunately, the generated path, assum-

ing minimal errors produced by the RNN, is already derived from a dynamic/kinematic formulation, with considerations to controllability enforced in the optimization. Under such, a control signal must exist that can track the system path accurately.

For this problem scenario, a simple feedback control loop is more than adequate for following the produced RNN state history. For this simple mechanical system, the velocity feedback portion of the control signal constitutes the error in desired velocity of the state with that of the RNN path, and the position feedback portion constitutes the error between the current state position and the desired position of the RNN path. This control signal is formulated as,

$$\mathbf{u}_{f,f}(x(t),\sigma(t_k)) = -K_p(\mathbf{x}_p(t)-\sigma_p(t_k)) - K_v(\mathbf{x}_v(t)-\sigma_v(t_k)), \tag{5.5}$$

where t is continuous time, t_k is sampled time per the RNN time interval, $\mathbf{x}_p$ is the position vector of the state, σ_p is the position vector of the RNN output path, $\mathbf{x}_v$ is the velocity vector of the state, σ_v is the velocity vector of the RNN output path, K_p is the position gain matrix, and K_v is the velocity gain matrix.

While the above controller design can maintain path tracking, no guarantees are provided with respect to constraint satisfaction if the RNN output path fails such. Preliminary results observed the RNN generated path as producing minor constraint violations over the training and validation solution sets. In order to combat this issue for real-time execution, localized potential functions are used about the current position state, derived from [57].

Provided local bounds $\mathbf{x}_{p,min,local}$ and $\mathbf{x}_{p,max,local}$ on the system position state at any

given time, potential functions of the form,

$$U(\mathbf{x}_p(t), \mathbf{x}_{p,s}) = \begin{cases} \frac{-c}{N_s \left\| \mathbf{x}_p(t) - \mathbf{x}_{p,s} \right\|} & \text{Eq. (3.1e)} \not\leq 0 \\ 0 & \text{otherwise} \end{cases} \tag{5.6}$$

are placed at uniform sample points $\mathbf{x}_{p,s}$ of resolution $\mathbf{r} < \left(\mathbf{x}_{p,max,local} - \mathbf{x}_{p,min,local}\right)$ about $\mathbf{x}_p(t)$. Fig. 5.1 provides a visualization of these functions with respect to an obstacle. In Eq. (5.6), N_s is a factor to mitigate the scaling issue when using multiple sample points, and c is a gain used for the controller. The condition of Eq. (3.1e) $\not\leq 0$ utilizes $\mathbf{x}_{p,s}$ instead of $\mathbf{x}_p$. The derivative of the repulsive potential functions results in the combined forces shown as,

$$\mathbf{F} = \sum_{s}^{N_s} \frac{-c\left(\mathbf{x}_p(t) - \mathbf{x}_{p,s}\right)}{N_s \left\| \mathbf{x}_p(t) - \mathbf{x}_{p,s} \right\|^2}. \tag{5.7}$$

The purpose of these potential functions is to provide local constraint satisfaction, not global satisfaction. As a result, egregious errors in the RNN path are not mitigated by the use of these functions. They simply serve as a means of preventing constraint violations on position in real-time and in a manner that could be employed locally about the agent.

As a result of the feedback controller and potential function forces, the executed controller results in the form,

$$\mathbf{u}_e = \mathbf{F}(t_k) + \mathbf{u}_{f,f}(\mathbf{x}(t), \sigma(t_k)), \tag{5.8}$$

where the term $\mathbf{F}(t_k)$ is calculated by Eq. (5.7) at each sampled time t_k. For application, gains of $c = 25$, $K_p = 40$, and $K_v = 40$ were used.

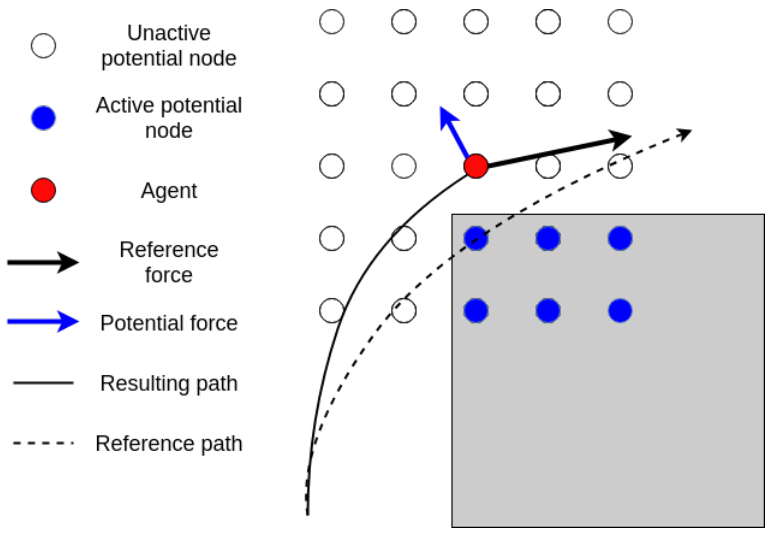

Figure 5.1: The use of localized potential functions can mitigate minor constraint errors of the state path produced by the RNN. Local potential nodes exist relative to the agent, which are activated when their position enters an object. Forces produced by the activated potential functions push the agent off from the reference trajectory to avoid constraint violations. The force produced by the reference trajectory prevents the local forces from driving the agent completely off the desired path.

5.2 Results

The performance of the methodology in a scenario is primarily dependent upon the closed-loop execution and $\sigma(t_k)$ path creation of the RNN (CL) compared against optimal paths and the controller execution over that path (CTRL) also compared to the optimal paths. Assessments for multiple training schemes were performed with visual comparisons and the root mean square error (RMSE) of a generated path compared against the optimized paths. The RMSE represents the square root of the loss function (mean square error) used in training and provides a metric for the average error between a generated path $\sigma(t_k)$ and an optimized path $G(t_k)$. The closer an RMSE value is to 0, the more accurate a prediction the RNN will produce for the given problem scenario.

Problem Scenario 1 provides a means of assessing this methodology's ability to produce near optimal paths (with respect to the optimization function in Eq. (5.4)) in cluttered environments. Results are produced to examine how well the RNN can recreate the desired optimal paths for training and validation data sets, as well as examining the use of the path-tracking controller on such. Three training schemes are utilized to judge the results with respect to normal training (no whole-path reinforcement training or autoencoder), training with the autoencoder, and training including both the autoencoder and the whole-path reinforcement training.

Training the RNN over the optimization sets G_{sets} without the autoencoder to represent the environment or the whole-path reinforcement training scheme produces fairly undesirable CL and CTRL results across both the training data sets and the validation data sets. This is obvious from the RMSE values provided in Table 5.1, which showcases high

error values in the position state with respect to the size of the domain explored (order of 10 meters). Velocity RMSE values are also high with respect to their domain values. An example CL output is displayed in Fig. 5.2 (left image) to provide the reader a sense of how the CL outputs under this training scheme behave with respect to the optimized solutions.

Inclusion of the autoencoder to compress point cloud representations of the obstacles increases the robustness of the represented environment, allowing for better training over the same set of data. The results of the autoencoder inclusion on the RMSE errors are also presented in Table 5.1. Comparing these values to those of the RNN trained in the normal training scheme showcase a marked improvement in the CL performance on the training data and on the validation data (approximately 30% reduction in training RMSE values and 35% reduction in validation RMSE values). The CL output as compared to the non-autoencoded RNN is shown in the right image in Fig. 5.2. While the path intersects the obstacle, the CL output better fits the optimized solution, an important step in improving training.

Finally, the use of whole-path reinforcement training further improves the results. Table 5.2 provides CL output training and validation RMSEs for the RNN training with the autoencoder and whole-path reinforcement training. For the training sets, the CL performance of the RNN under whole-path reinforcement training results in significant reductions to the position and velocity RMSE compared to just the use of the autoencoded environment (up to 60% further reductions). The left image of Fig 5.3 provides the resulting CL path under this training scheme of the scenario shown in Fig. 5.2. Unfortunately, these improvements do not extend as well to the validation set RMSE values.

No autoencoded env.	Training data	Validation data
RNN CL position (m) RMSE	2.59	3.80
RNN CL velocity (m/s) RMSE	1.16	1.40
Autoencoded env.	Training data	Validation data
RNN CL position (m) RMSE	1.76	2.34
RNN CL velocity (m/s) RMSE	0.81	0.98

Table 5.1: RMSE path values of RNN in closed-loop (CL) execution over both training and validation data sets for training scheme without autoencoded environ. RMSE values are provided in the units used for the property stated. The closer an RNN's RMSE value is to zero, the more accurate its ability is to track the desired optimal solution. The normal training scheme provides poor results when comparing the RMSE magnitudes to that of the domain the pos

Comparatively, the use of just the autoencoded environment provides validation RMSEs on the same order as the combined training, if not slightly better. A primary reason for this result is the increased amount of training performed on just the training data sets when utilizing the whole-path reinforcement training. As a result, information pertaining to a select amount of optimized solutions is repeatedly observed, leading to overfitting of the RNN to these sets. Larger training and validation data sets can help alleviate this issue, but further research is required to determine how to better extend the patterns learned in the whole-path reinforcement training to the validation training sets.

A further assessment of this methodology's application to the obstacle avoidance scenario includes examination of the path-tracking control's output path (CTRL output) and control signals as compared to the optimal control solution. The controller's performance on the RNN CL path is a vital component to the assessment of this methodology as it represents the end product, since the online execution informs how well the RNN CL

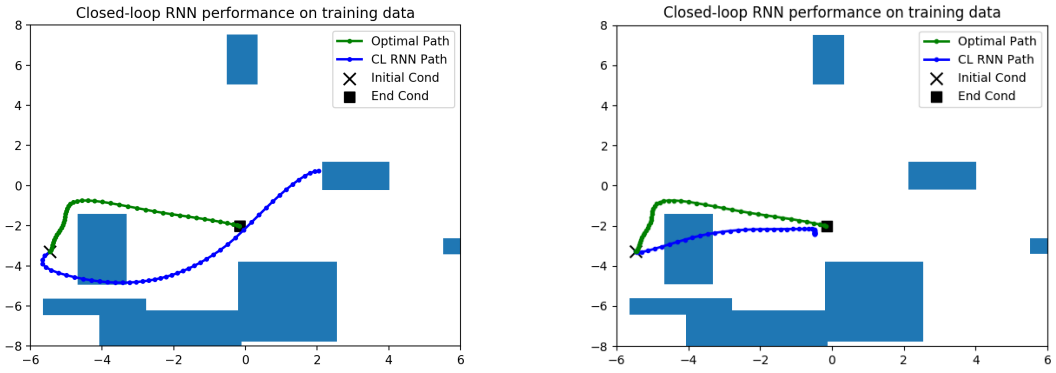

Figure 5.2: Example of the RNN closed-loop (CL) path compared against the optimized solution from the training set with no autoencoded environment (left) and with autoencoded environment (right). The normal training scheme without an autoencoded environment produces larger errors with respect to the desired state path than compared to the RNN result using the autoencoded environment. While the autoencoded version intersects the obstacle compared to the non-autoencoded version, the form of the desired optimal solution is better understood, an important step for improving training.

paths can actually be utilized and what adjustments may need to be made. The CTRL position, velocity and control signal RMSE values are provided in Table 5.2. As observed, the RMSE values of the CTRL paths nearly match identically those of the CL paths. This indicates the ability of the path-tracking controller to maintain the desired output path in time. An example of the CTRL path is provided in the right image in Fig. 5.3, which showcases a stronger fit to the desired optimal solution than the CL path in part due to the constraint violation mitigations through the potential function controller. Additional examples of CTRL results on validation data sets are provided in Fig. 5.4. These validation examples represent the RNN's ability to extend the solution space. From the images, assessments can be made (detailed in the Fig. 5.4) that the RNN learned the solution space fairly well, but not enough so that controller mitigations of constraint violations

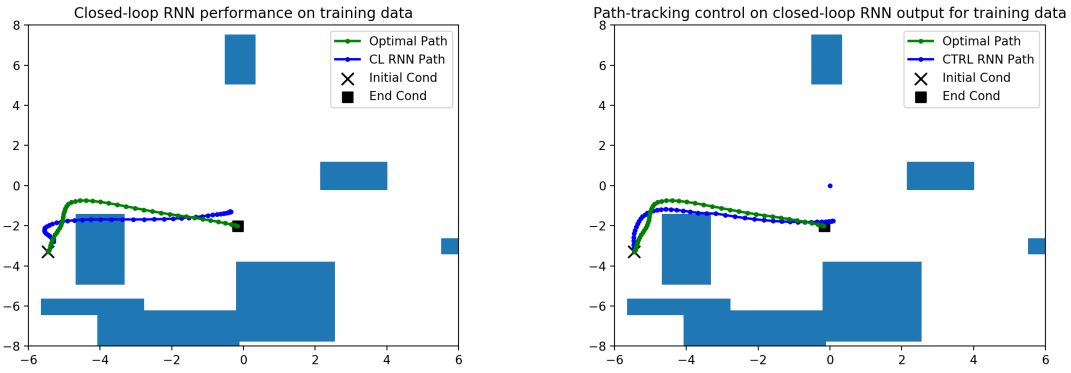

Figure 5.3: Training example RNN CL output utilizing full training scheme (autoencoder + whole-path reinforcement). As compared to the figures in Fig. 5.2, the CL output path more closely fits the form of the desired optimal solution. Minor constraint violations are still present, though, but less severe than the result from the use of just the autoencoded environment.

are uncommon. These path violations from the RNN CL output impact the performance of not only the CTRL paths, but also the CTRL control signals in respect to the desired optimal control signals.

The CTRL control signal RMSE values are much greater in comparison to their respective domain magnitudes (order of 10 N) than those of the CTRL position and velocity values. Fig. 5.5 provides a comparison of the x-axis control signals with respect to the optimized control signals for the validation examples in Fig. 5.4, from which a few assessments can be made. Control spikes in the top left and bottom right images correspond to activations of the local potential functions. Higher density potential grids would smooth out such spikes. In general, the control signals from the CTRL paths begin with a large offset followed by a trending behavior towards the optimal control signal. This large offset is due to the common initial error prediction by the RNN at initialization of the paths. Improvements to the CL paths generated by the RNN would lead to more

	Training data	Validation data
RNN CL position (m) RMSE	0.53	2.52
RNN CL velocity (m/s) RMSE	0.36	1.00
CTRL position (m) RMSE	0.48	2.54
CTRL velocity (m/s) RMSE	0.40	0.98
CTRL control signal (N) RMSE	8.30	13.98
CTRL optimization metric (Ave. $\lvert\%$ Error$\rvert$)	84%	85%

Table 5.2: RMSE path values of RNN in closed-loop (CL) execution over both training and validation data sets for training scheme without autoencoded environ. RMSE values are provided in the units used for the property stated. The closer an RNN's RMSE value is to zero, the more accurate its ability is to track the desired optimal solution. The normal training scheme provides poor results when comparing the RMSE magnitudes to that of the domain the pos

accurate control signals. Furthermore, the average percent error of the optimization metric is also provided in Table 6.2. Again, further path improvements by the RNN would decrease this error.

The above results showcase the ability of an RNN to learn the solution space of an optimization problem for a simple dynamic system maneuvering a cluttered environment. The inclusion of an autoencoder for the environment and whole-path reinforcement training greatly improve the CL results of the RNN on the training data, while generating less of an improvement on the validation data. For all of the produced CL paths, RNN generation only required on the order of 10's of milliseconds, orders of magnitude less than the time required to generate a single optimal solution from the kinodynamic RRT*. Problem Scenario 1 highlights the methodologies capabilities in satisfying varied environment constraints well showcasing clear areas for improvement.

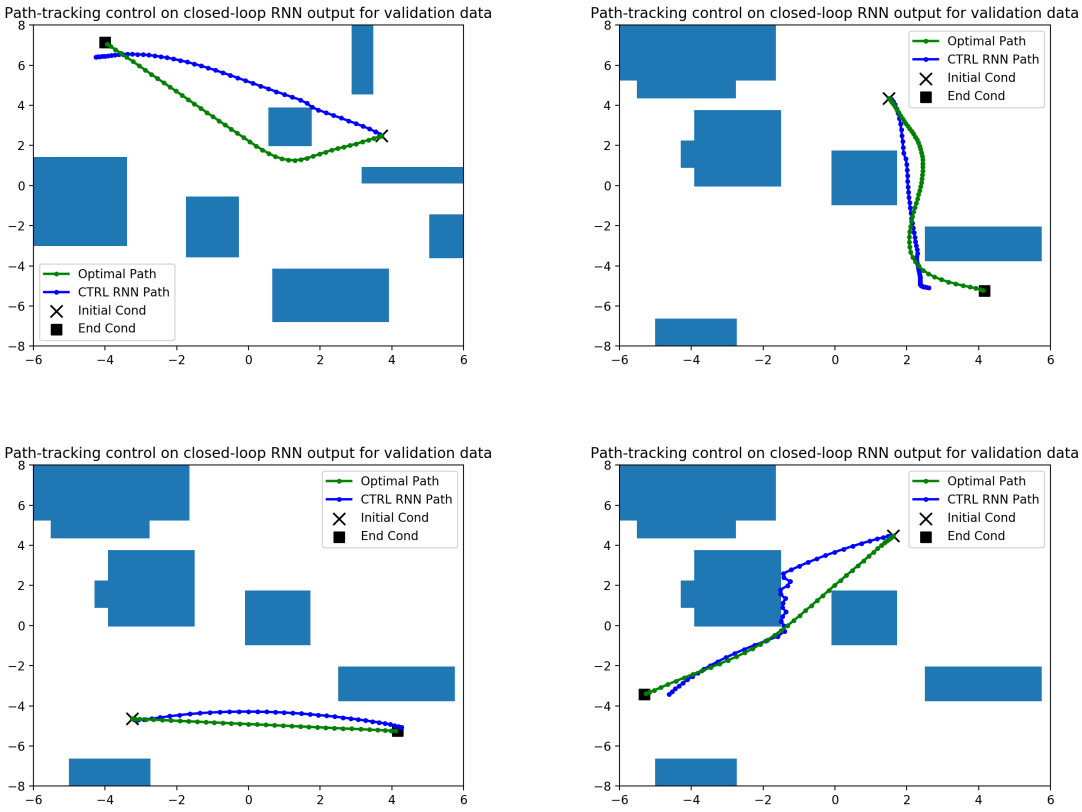

Figure 5.4: Validation data set examples of resulting CTRL path over the generated RNN CL paths. The top left and bottom left images represent desired results, while the top right and bottom right images represent undesired results. Top left showcases the RNN generalizing solutions, with the controller mitigated only minor constraint violations. Bottom left showcases a standard result operating outside of the obstacle environment. Top right showcases strong generalization in avoiding obstacle collisions while not meeting the end condition. Bottom right showcases major constraint violation, requiring the executed controller to mitigate the such. The oscillating path is due to a sparse potential function grid about the agent updating at a lower frequency than the feedback controller signal (10 Hz vs 100 Hz).

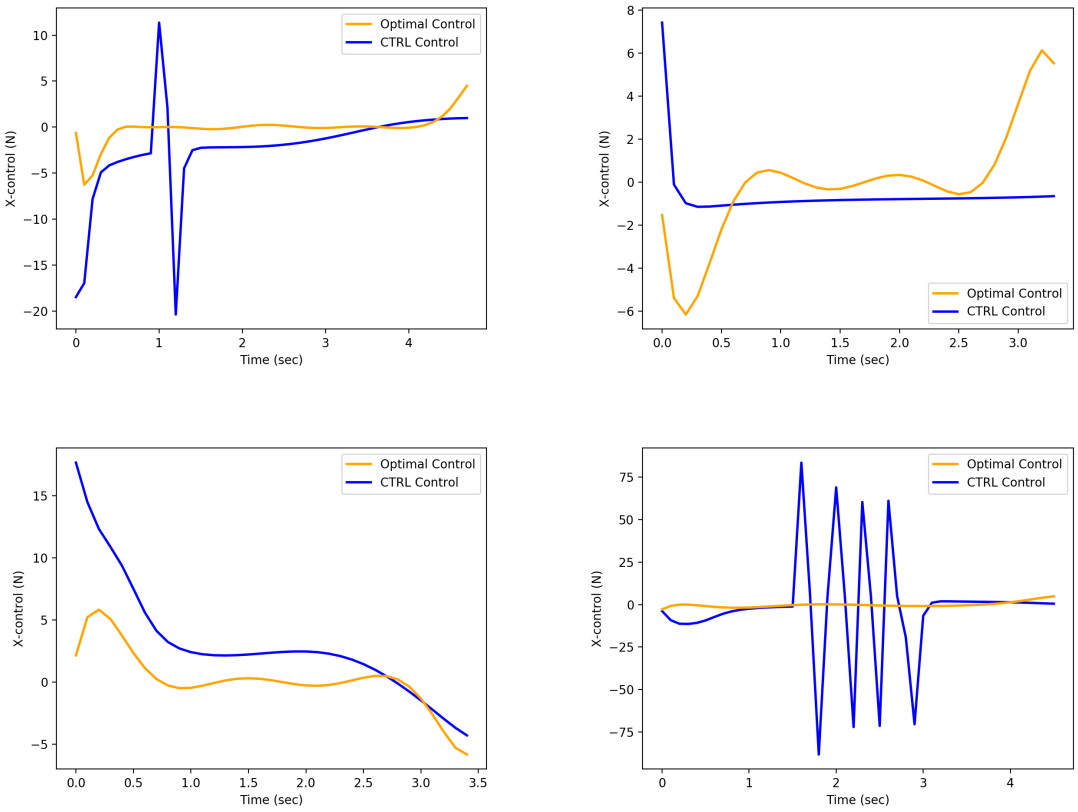

Figure 5.5: X-axis control signals corresponding to the validation examples provided in Fig. 5.4 compared against the optimal control signals. Control spikes in the top left and bottom right images correspond to activations of the local potential functions. Higher density potential grids would smooth out such spikes. In general, the control signals from CTRL paths begin with a large offset followed by a trending behavior towards the optimal control signal. This large offset is due to the common initial error prediction by the RNN at initialization of the paths. Improvements to the CL paths generated by the RNN would lead to more accurate control signals.

Chapter 6: Problem Scenario 2: 2D Multi-agent Synchronized Rendezvous and Collision Avoidance

To examine the effectiveness of the methodology proposed in this thesis with respect to higher dimension systems, a multi-agent synchronized rendezvous and collision avoidance problem is examined. The problem formulation and application of the methodology with respect to it is further described in the following sections.

6.1 Implementation

The optimization formulation of Chapter 3 presented in the methodology places no restrictions on the number of agents. With respect to problems involving multiple agents, this formulation equates to a centralized planner. To account for multi-agent systems, the

optimization formulation was modified to produce the following scheme:

$$\underset{\mathbf{u}(t),t_f}{\text{minimize}} \quad \begin{array}{l} \int_0^{t_f} \left(\sum_{i=1}^{N_a} \mathscr{W}(t,\mathbf{x}_i(t),\mathbf{u}_i(t)) \right) dt + \\ \sum_{i=1}^{N_a} \mathscr{L}(\mathbf{x}_i(0),t_f,\mathbf{x}_i(t_f)) \end{array}, \tag{6.1a}$$

subject to

$$\dot{\mathbf{x}}_i = \mathbf{f}(\mathbf{x}_i(t),\mathbf{u}_i(t)), \tag{6.1b}$$

$\forall i \in \{1,...,N_a\}$

$$\mathbf{x}_i(0) = \mathbf{x}_{i,0}, \tag{6.1c}$$

$$\mathbf{x}_i(t_f) = \mathbf{x}_{i,f}, \tag{6.1d}$$

$$\mathbf{C}_i(\mathbf{x}(t),\mathbf{u}(t),\mathbf{P}_i) \leq \mathbf{0}. \tag{6.1e}$$

In this formulation, the index i corresponds to an individual agent and its respective parameters. As stated before, this format represents a centralized planner among all agents. Because of such, two RNN architectures were of particular interest for study, a centralized and a decentralized version. The centralized version corresponds to a single RNN accounting for all agents, while the decentralized RNN is constructed for each individual RNN with limited knowledge on the rest of the agents. Investigation of the two approaches used the following problem formulation.

Each agent is modeled as a point mass subject to linear dynamics,

$$\dot{\mathbf{x}}_i = \mathbf{v}_i \tag{6.2}$$

$$\dot{\mathbf{v}}_i = \mathbf{u}_i m, \tag{6.3}$$

where $m = 1$. The state boundary is $(-6,-8) \leq (x,y) \leq (6,8)$ and $(-2.5,-2.5) \leq (v_x,v_y) \leq (2.5,2.5)$, with control constraints of $(-10,-10) \leq (u_x,u_y) \leq (10,10)$. The

environment consists of collision avoidance between all agents, with $\mathbf{C}_i$ formulated as,

$$0.3 - \left\|\mathbf{x}_{p,i} - \mathbf{x}_{p,j}\right\| \quad \forall j \in \{i, ..., N_a\}, \tag{6.4}$$

where x_p represents the position of an agent. The desired final time is fixed at 20 seconds. Initial state positions (with zero velocity) are sampled randomly within the state domain, while final state positions (with zero velocity) are set at equal intervals along the x axis, fixed for each agent. The optimization function to minimize is,

$$\int_0^{t_f=20} \sum_{i=1}^{N_a} \|\mathbf{u}_i\| \, dt. \tag{6.5}$$

The above formulation allowed for direct solutions using the NLP solver and ignoring the kinodynamic RRT*. Utilizing 10 agents ($N_a = 10$), approximately 5,000 solutions were generated for training and validation (70/30 split), each consisting of randomized initial conditions for each agent. The solver SNOPT [51] was utilized for solving the optimization problem formulated as an NLP problem in each configuration. The centralized RNN was constructed with 5 hidden layers (sizes 300, 200, 150, 100, and 80) utilizing the hyperbolic tangent activation function and an output layer utilizing a linear activation. The decentralized RNNs were constructed with 4 hidden layers (sizes 150, 110, 70, 20, and 4) utilizing the hyperbolic tangent activation function and output layers utilizing the linear activations, too. Keras [52] with the TensorFlow [53] backend were utilized for network construction and training through the Nesterov Adam optimization scheme [54] (learning rate of 0.002 and schedule delay of 0.01). The path-tracking controller for each agent consisted of a simple feedback control law (Eq. (5.5)) over the reference track. Controller execution utilized gains $K_p = K_v = 25$ for all agents.

6.2 Results

The multi-agent problem scenario is designed to assess the performance of both the centralized and decentralized RNN forms in recreating synchronized optimal paths with collision avoidance constraints. In the centralized form, all agent states are known to all other agents states within the network. In the decentralized form, only initial global information (the initial positions of all agents) is provided to each RNN. The performance of both RNNs is assessed on their ability to recreate the optimal state paths over training and validation solutions. Furthermore, the agent controllers are executed over each path to assess the performance in following the desired path and recreating the optimized control signal, tied directly to the optimization metric defined for the problem.

The root mean square error (RMSE) of an RNN's state output ($\sigma(t_k)$) in closed-loop (CL) form against the training and validation data is used in assessing the overall performance of both networks. Table 6.1 provides a comprehensive overview of both the centralized and decentralized RNN RMSE performances on the training and validation sets. Note that the RMSE value of the decentralized RNN is calculated across all agents.

Fig. 6.1 and Fig. 6.2 provide training and validation examples (respectively) of the CL RNN outputs as compared to the optimal paths. When examining both the RMSE values and example plots, it becomes obvious that the decentralized RNN outperforms that of the centralized RNN. This is observable in Fig. 6.1 and Fig. 6.2 in not only the greater drift present in the CL paths of the centralized controller but also the unaligned nodes of the produced path (representing evenly space points in time).

Interestingly, though, while the decentralized RNN may outperform that of the cen-

	Training data	Validation data
Cent. RNN CL position (m) RMSE	0.520	0.694
Cent. RNN CL velocity (m/s) RMSE	0.384	0.537
Decent. RNN CL position (m) RMSE	0.310	0.557
Decent. RNN CL velocity (m/s) RMSE	0.278	0.491

Table 6.1: RMSE path values of centralized and decentralized RNNs in closed-loop (CL) execution over both training and validation data sets. RMSE values are provided in the units used for the property stated. The closer an RNN's RMSE value is to zero, the more accurate its ability is to track the desired optimal solution. Comparatively, the decentralized RNN outperformed that of the centralized RNN.

tralized version in the overall path accuracy, both sets were consistent in ending at the desired final locations within the execution time of 20 seconds. Adversely, while the whole-path reinforcement training scheme does well in aligning the state paths overall, it did tend to produce initial path sequences that diverged before aligning back with the optimized path, a point observed in the *Results* section of Chapter 5. This notion along with the networks' ability to consistently hit the desired final location indicate that the network is able to generalize the optimization problem through time and needs further training improvements to increase the CL path accuracy.

The validation sets for both networks showcases the ability of them to generalize over new agent configurations. When comparing the decentralized RMSE values to the centralized RMSE values on the validation set, though, we observe a less obvious improvement. This indicates that both RNN setups may be generalizing the solution space in a more similar manner, one that may become more apparent with a larger data set.

Controller executions were performed over all closed-loop RNN paths for compar-

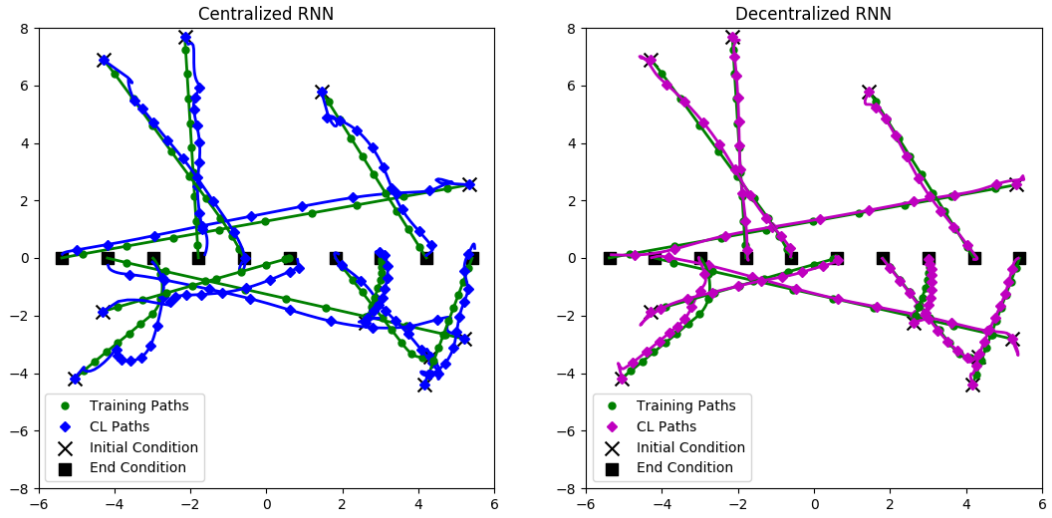

Figure 6.1: Training example of the centralized and decentralized closed-loop (CL) state outputs compared against the optimized path. Nodes are added at equal intervals (2 secs) to represent fixed-interval points in time. Diamond nodes represent CL paths and circle nodes represent the optimized path. X's represent the initial conditions of all agents, and squares represent the end conditions. Generally, the decentralized RNN produces more accurate paths than the centralized version, while both consistently end at the desired final condition.

ison against the optimal state and control signals. Table 6.2 displays the RMSE executed control (CTRL) values of the resulting state paths, control signals, and evaluation of the optimization metric.

Fig. 6.3 and Fig. 6.5 provide training and validation examples of the resulting state paths from the controller execution. Fig. 6.4 and Fig. 6.6 provide the controller signals association with the training and validation examples of Fig. 6.3 and Fig. 6.5. The paths followed by all agents in the controller executed form tend to follow that of the closed-loop paths produced by the RNNs themselves, resulting in the same comparisons between the centralized and decentralized RNNs. Greater RMSE values are present in the CTRL cases as compared to the CL cases. This appears influenced by the greater

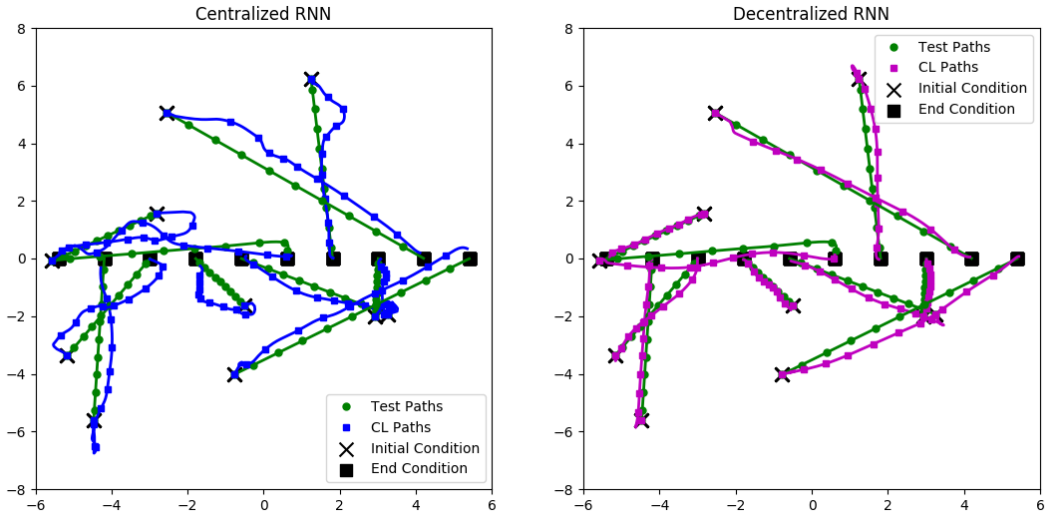

Figure 6.2: Validation example of the centralized and decentralized closed-loop (CL) state outputs compared against the optimized path. Nodes are added at equal intervals (2 secs) to represent fixed-interval points in time. Diamond nodes represent CL paths and circle nodes represent the optimized path. X's represent the initial conditions of all agents, and squares represent the end conditions. Generally, the decentralized RNN produces more accurate paths than the centralized version, while both consistently end at the desired final condition.

initial lag introduced by the greater initial errors present within the CL paths. The path-tracking controller of an agent spends more time around the initial condition before being yanked along the path, resulting in a greater general error between the CTRL path and the optimized path.

Unfortunately, this translates to less than favorable control signals observed in Fig. 6.4 and Fig. 6.6 and the produced RMSE values in Table 6.2. While the control signals trend the same path, the resulting integration of the optimization metric (i.e. the control norm) results in more excessive mean errors when compared to the optimal solutions. Observable in both graphs is the common control spike present around the 0.1 second mark. This is tied to the higher error in the initial closed-loop state output of the RNNs

	Training data	Validation data
Cent. CTRL position (m) RMSE	0.777	0.912
Cent. CTRL velocity (m/s) RMSE	0.577	0.587
Cent. CTRL control (N) RMSE	0.862	0.883
Cent. CTRL integrated control (Ave. $\lvert\%$ Error$\rvert$)	61.3	62.3
Decent. CTRL position (m) RMSE	0.637	0.760
Decent. CTRL velocity (m/s) RMSE	0.554	0.567
Decent. CTRL control (N) RMSE	0.819	0.850
Decent CTRL integrated control (Ave. $\lvert\%$ Error$\rvert$)	61.4	62.4

Table 6.2: RMSE path and control values of executed control (CTRL) over the centralized and decentralized RNNs for both training and validation data sets. RMSE values are provided in the units used for the property stated. The closer an RNN's RMSE value is to zero, the more accurate its ability is to track the desired optimal solution. Comparatively, the decentralized RNN outperformed that of the centralized RNN.

observed in the closed-loop graphs. With better training, this initial error can be reduced, producing smoother closed-loop state paths and better resulting control trends.

Problem Scenario 2 showcases the methodology's application to a higher dimensionality problem involving multiple agents. CL path performance utilizing the whole-path reinforcement scheme performed similarly to the results of Chapter 5 involving the obstacle avoidance problem. Similarly, results indicate that improvements to the whole-path training scheme as well as better extension to validation data will improve the CTRL performance. For this specific application, decentralizing the RNNs helped improve results, showcasing the extension of the solution space formed in a centralized manner to a decentralized application. Further research is required to assess this methodology's per-

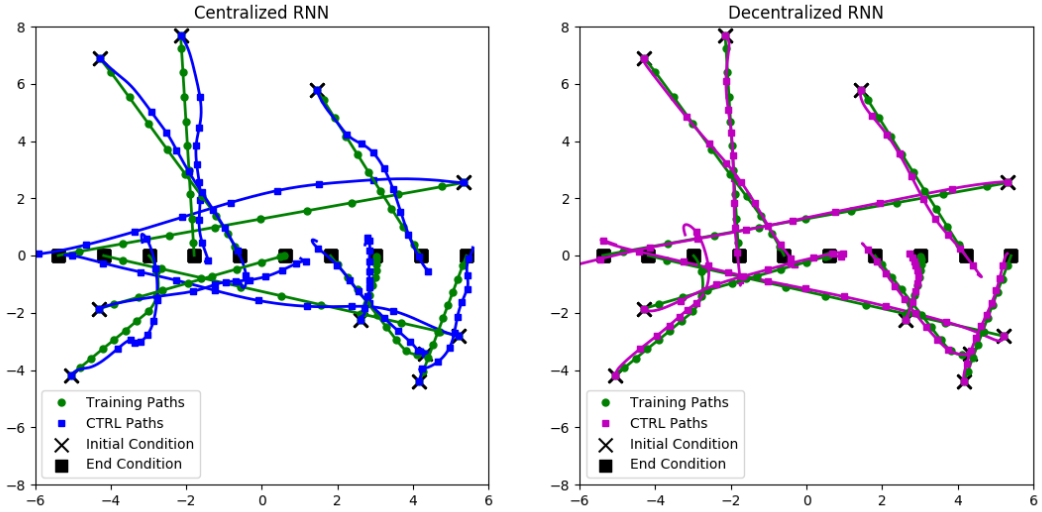

Figure 6.3: Training example of the centralized and decentralized executed control (CTRL) state outputs compared against the optimized path. Nodes are added at equal intervals (2 secs) to represent fixed-interval points in time. Diamond nodes represent CTRL paths and circle nodes represent the optimized path. X's represent the initial conditions of all agents, and squares represent the end conditions. Generally, the decentralized RNN produces more accurate paths than the centralized version, while both consistently end at the desired final condition.

formance in even higher dimensionality systems, approaching that of swarms. Currently, the prohibiting factors appear to be the use of the kinodynamic RRT* and NLP optimization for generating centralized optimization data, which does not scale well.

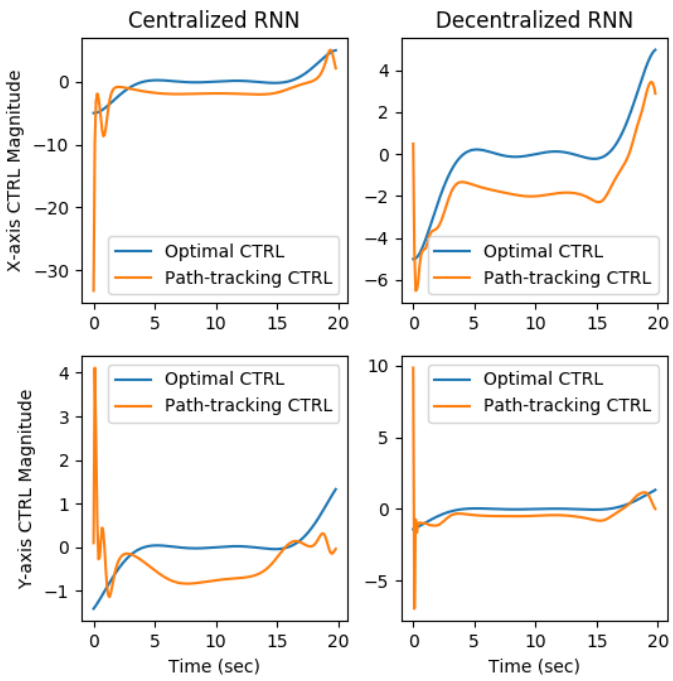

Figure 6.4: Control signal outputs for agent 1 when following state paths produced by the centralized and decentralized RNNs on the training example. In the ideal performance, the path-tracking control trends that of the optimal control. Initial errors in the CL paths produce the initial spikes in the control signals.

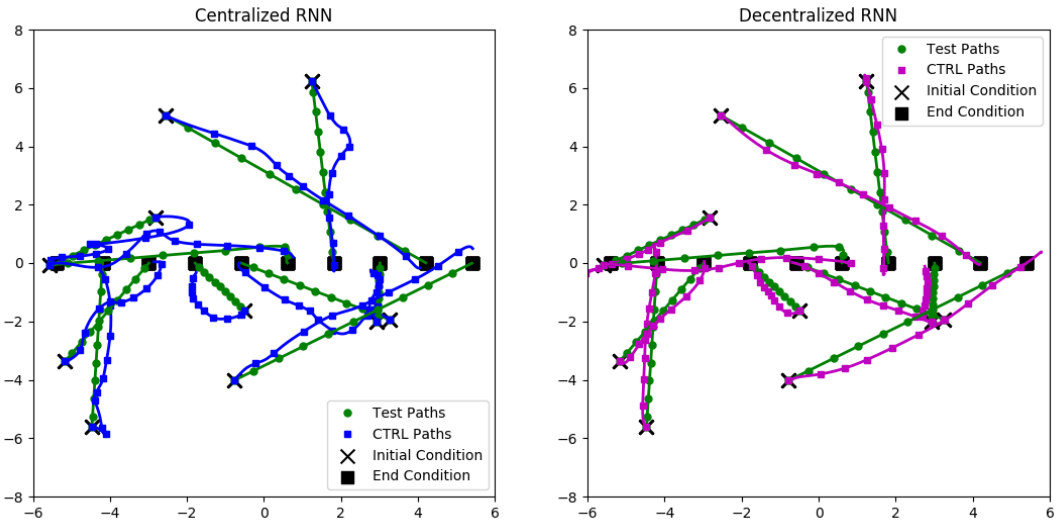

Figure 6.5: Validation example of the centralized and decentralized executed control (CTRL) state outputs compared against the optimized path. Nodes are added at equal intervals (2 secs) to represent fixed-interval points in time. Diamond nodes represent CTRL paths and circle nodes represent the optimized path. X's represent the initial conditions of all agents, and squares represent the end conditions. Generally, the decentralized RNN produces more accurate paths than the centralized version, while both consistently end at the desired final condition.

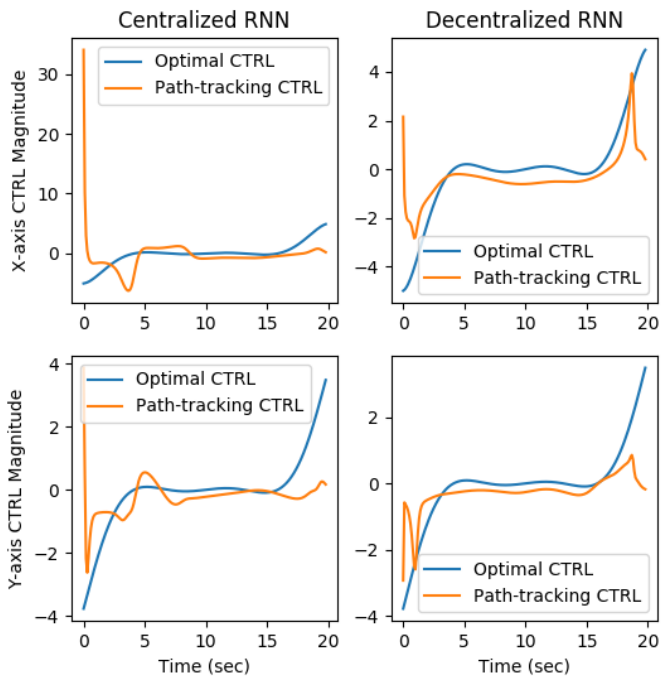

Figure 6.6: Control signal outputs for agent 1 when following state paths produced by the centralized and decentralized RNNs on the validation example. In the ideal performance, the path-tracking control trends that of the optimal control. Initial errors in the CL paths produce the initial spikes in the control signals.

Chapter 7: Problem Scenario 3: Actuated Double Pendulum

To examine the effectiveness of the methodology proposed in this thesis with respect to highly nonlinear systems, an actuated double pendulum problem is examined. The problem formulation and application of the methodology with respect to it is further described in the following sections.

7.1 Implementation

The previous two scenarios each explored linear dynamical systems, focusing on issues related to environment definitions and state space size. Problem Scenario 3 investigates the methodology's capabilities in regards to complex nonlinear system solutions, specifically in regards to the capabilities of an RNN to fit optimized solutions of a nonlinear system.

The dynamics of an actuated double pendulum system, with joint angles θ_1 and θ_2, joint velocities $\dot{\theta}_1$ and $\dot{\theta}_2$, lengths l_1 and l_2, and endpoint masses m_1 and m_2, are,

$$M\ddot{\theta} + C\dot{\theta} + G = \tau, \tag{7.1}$$

$$M = \begin{bmatrix} (m_1 + m_2)l_1^2 & m_2 l_1 l_2 \cos(\theta_1 - \theta_2) \\ m_2 l_1 l_2 \cos(\theta_1 - \theta_2) & m_2 l_2^2 \end{bmatrix}, \tag{7.2}$$

$$C = \begin{bmatrix} 0 & m_2 l_1 l_2 \sin(\theta_1 - \theta_2) \\ -m_2 l_1 l_2 \sin(\theta_1 - \theta_2) & 0 \end{bmatrix}, \tag{7.3}$$

$$G = \begin{bmatrix} l_1(m_1 + m_2)g\sin(\theta_1) \\ l_2 m_2 g \sin(\theta_2) \end{bmatrix}, \tag{7.4}$$

where g is standard gravity.

For the scenario explored, $m_1 = m_2 = 4kg$ and $l_1 = l_2 = 2m$. The path planning problem in this scenario involves actuating the double pendulum from a zero angle rest configuration (i.e. $\theta = \mathbf{0}$ and $\dot{\theta} = \mathbf{0}$) to a final end-effector rest position $x = 0$ and $0 < y < 3.5$. This final end-effector position (x, y) corresponds to an angle configuration (for $l_1 = l_2$) of $\theta_f = \left(\pi - \arccos(\frac{y}{2l_2}), \pi + \arccos(\frac{y}{2l_2})\right)$ and $\dot{\theta}_f = (0,0)$. Therefore, no explicit environment is defined besides the final state value, represented as $x_f = \theta_f$ in the optimization formulation. Furthermore, the states were restricted as $(0.0, 0.0, -\pi/2, -\pi/2) \leq (\theta, \dot{\theta}) \leq (3\pi/2, 3\pi/2, \pi/2, \pi/2)$. The chosen optimization function to minimize was the mechanical energy,

$$\int_0^{t_f} (\tau \cdot \dot{\theta})^2 dt. \tag{7.5}$$

Similar to the previous scenarios, the kinodynamic RRT* was utilized for generating over 500 solutions (80/20 split between training/validation) for varied final end-

effector heights. The RNN architecture consisted of 5 hidden layers (sizes 100/80/60/60) with the hyperbolic tangent activation function and a linear output layer. ADAM was again used for training with a learning rate of 0.001. No path-tracking controller was explored, and instead the closed-loop performance of the RNN on the training and validation sets was primarily examined.

7.2 Results

Problem Scenario 3 is presented to assess the capabilities of an RNN in learning optimized solutions for highly nonlinear systems, specifically actuating a double pendulum from rest up to a final resting configuration. This scenario presents a particularly difficult optimization solution due to the initial "wiggle" of the pendulum before swinging up into the final configuration, showcased in an optimal solution example of Fig. 7.1.

The CL performance of the RNN is presented for a training scheme without whole-path reinforcement training and that with it. The RMSE values of the RNN CL output for both training and validation data sets are displayed in Table 7.1.

Readily apparent is the significant impact that whole-path reinforcement learning has on the CL output for the training data sets. The CL output RMSE is reduced by a factor of 70% for the angular position and by 60% for the angular velocity on the training data. This reduction is not as significant for the validation data sets, which barely approach 20% reductions in either the angle or angular velocity. As previously mentioned in the results of Chapter 5 and 6, the whole-path training scheme provides strong improvements to training examples and their CL executions, but further modifications are required

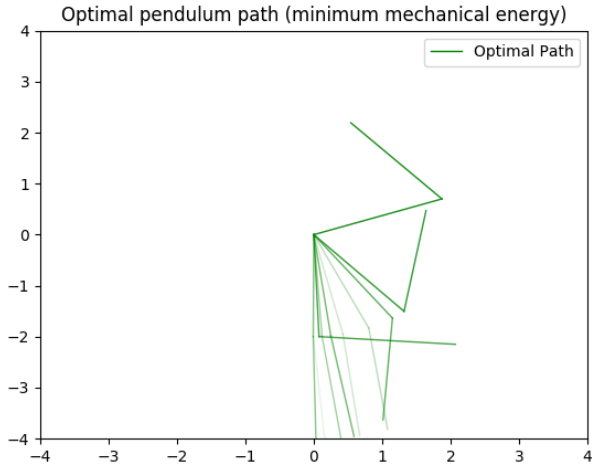

Figure 7.1: Example optimization solution for pendulum swinging from initial zero condition to a final configuration in which the the end effector is at rest at $(x,y) = (0,2.3)$. Opaqueness increases in time. Optimization of the mechanical energy results in a "wiggle" pattern at the start, which flicks the lower pendulum upwards before the entire system swings up. This initial behavior provides a difficult optimization pattern to learn.

to extend this learning to the validation data sets. In this case involving a highly nonlinear system, this plays a significant hurdle in the effectiveness of using these generated paths for path-tracking control of the pendulum to the desired final condition.

Fig. 7.2 provides a visual of the whole-path reinforcement training's impact on the RNNs ability to learn the training sets. Provided are an ideal case (top images) and a less than ideal case (bottom images). In both cases, the angular values arrive at the desired final angle, approaching in the correct manner. Fig. 7.3 showcases the Cartesian position CL paths of the pendulum arm with respect to the optimal paths through time. Fig. 7.4 showcases validation examples of the CL angular paths in time. As mentioned, the path disparities between the CL paths and optimal solutions showcases the whole-path reinforcement training's weakness in extending its capabilities to the validation set.

No reinforcement training	Training data	Validation data
RNN CL angle (rad) RMSE	1.21	1.24
RNN CL angular rate (rad/s) RMSE	1.64	1.68
Reinforcement training	Training data	Validation data
RNN CL angle (rad) RMSE	0.37	1.00
RNN CL angular rate (rad/s) RMSE	0.65	1.33

Table 7.1: RMSE values are provided for the RNN CL output against the optimized solutions for the training data set and the validation sets when using the whole-path reinforcement training and when not using it. For this nonlinear system, the whole-path reinforcement training provides significant CL performance increases on the training data, but not as significant increases on the validation data.

The application of the first two steps in the methodology to a highly nonlinear system and optimization solution space showcase the required improvements needed to the whole-path reinforcement training scheme, specifically with respect to its generalization for validation data sets. Otherwise, improvements of the CL output for the training data were apparent, showcasing the ability of the RNN to learn the desired solution structure.

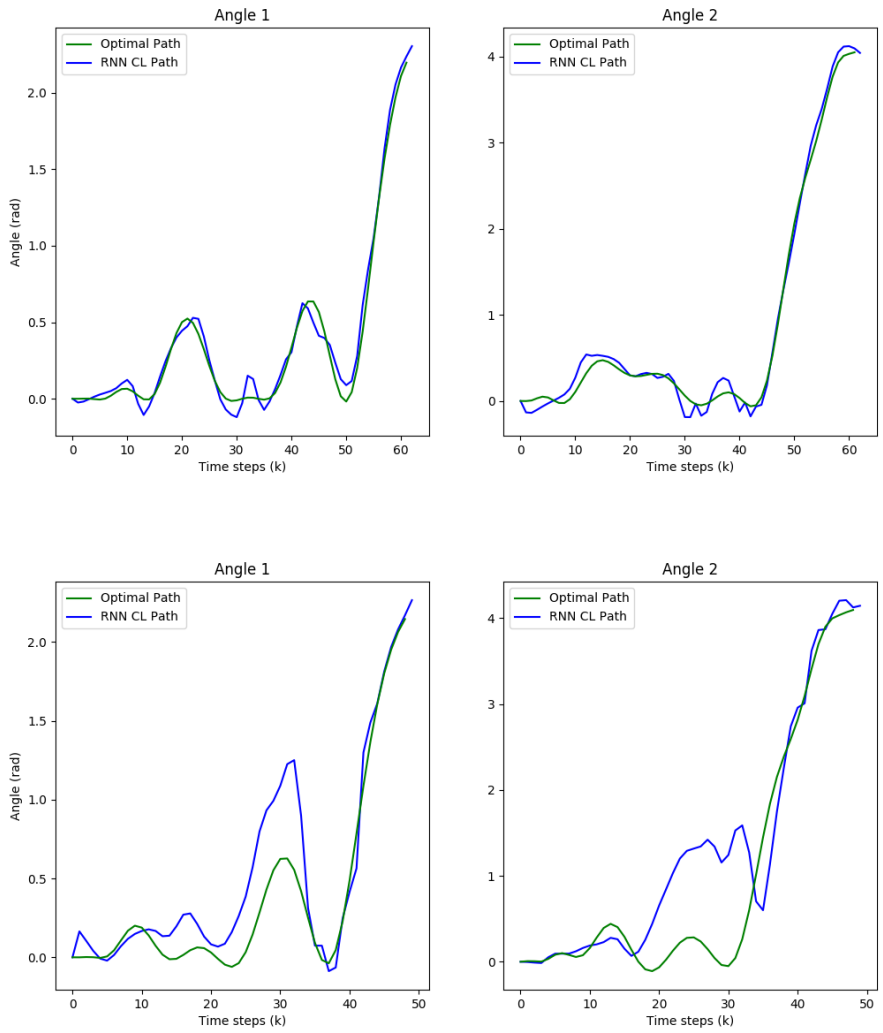

Figure 7.2: The two sets of images showcase a training data example of a strong CL output (top image) and a weaker one (bottom image) as compared to the optimal solution paths. The left images correspond to the first angle θ_1, and the right images correspond to the second angle θ_2. Both output sets showcase strong pattern matching of the end of the path, common across most training examples.

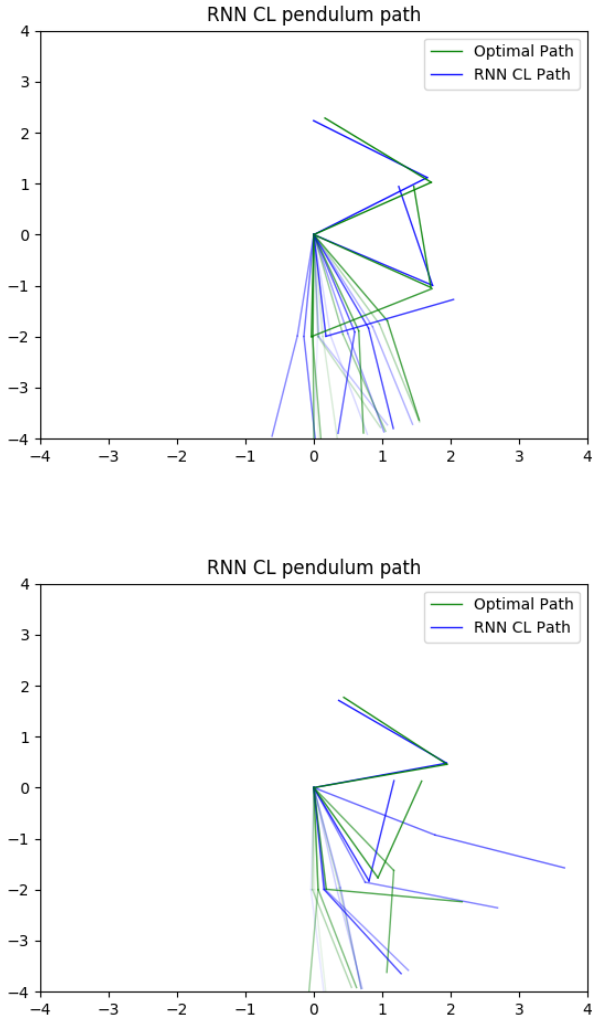

Figure 7.3: The top and bottom images correspond to the top and bottom images of Fig. 7.2, respectively. These showcase the Cartesian pendulum paths of the optimal solution and the fitted RNN CL output. Opaqueness of the pendulum rods increases with time.

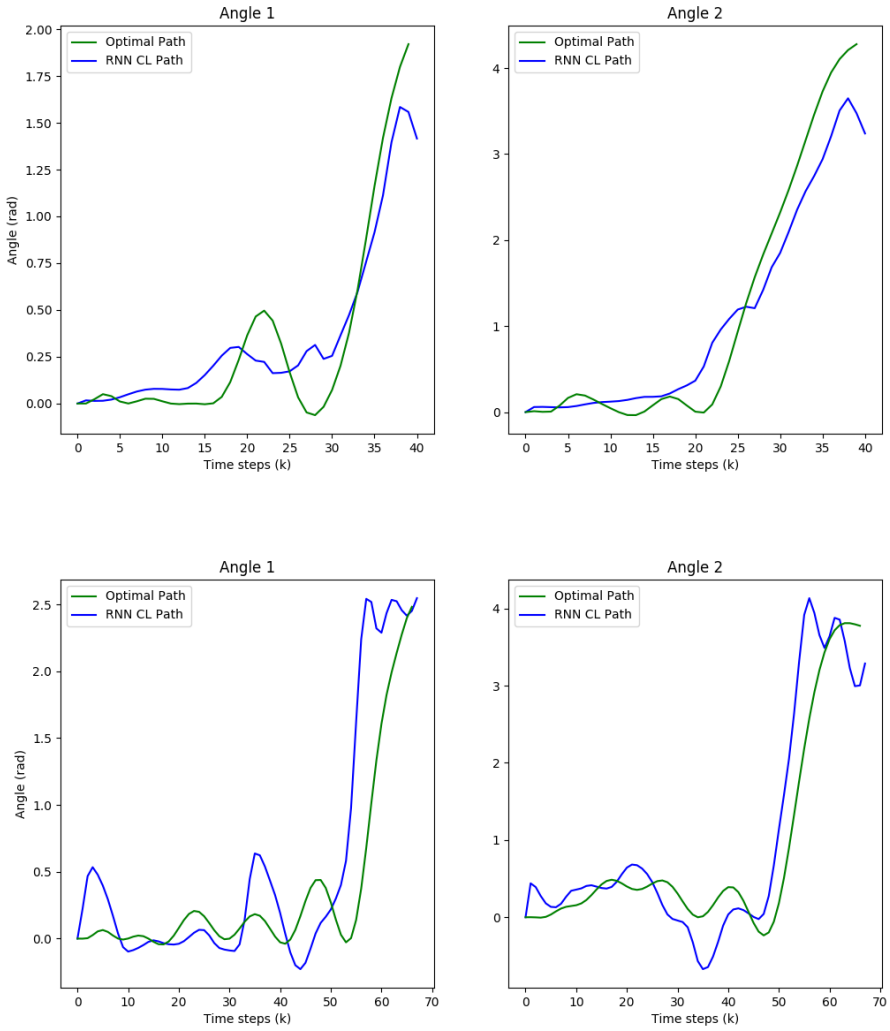

Figure 7.4: The two sets of images showcase a validation data examples of the CL output compared to the optimal paths. The left images correspond to the first angle θ_1, and the right images correspond to the second angle θ_2. The validation examples trend the optimal solutions, but do not accurately match such.

Chapter 8: Bounded Set Propagation of Recurrent Neural Networks

Bounded set propagation (a type of reachability analysis) of the recurrent states in path planning RNNs provides a means for bounding the total set of possible outputs for a given set of inputs to an RNN. Propagating these sets through the entire execution of a path planning RNN allows for analysis on the entire reachable sets of states that the RNN can produce. While not integrated into the methodology directly, the contents of this chapter explore improvements to the formulations provided in [58], primarily due to an observed explosion of the bounded sets when applied to Problem Scenario 1. These improvements include tighter approximations and faster computations.

$$\begin{aligned} \mathbf{x}_t &= \Phi(\mathbf{x}_{t-1}, \mathbf{u}_t), \\ \text{where } \Phi(\cdot) &= \sigma_L \circ \sigma_{L-1} \circ \ldots \sigma_1(\cdot), \\ \text{with the form } \sigma_l(\cdot) &= \sigma_l(W_l \mathbf{h}_l + \mathbf{b}_l) \end{aligned} \tag{8.1}$$

The bounded set propagation formulation begins with the NARX RNN form defined in Eq. (8.1), where W_l and $\mathbf{b}_l$ are network weights and biases per layer, $\mathbf{h}_1$ is the network input composed of recurrent state $\mathbf{x}_{t-1} \in \mathbb{R}^N$ and environment input $\mathbf{u}_t \in \mathbb{R}^{O+2N}$ (consisting of environment P, initial state x_0, and final state x_f), and σ_L corresponds to output $\mathbf{x}_t \in \mathbb{R}^N$. Next, the input set X_{t-1} for propagation is defined by Eq. (8.2), where

$\underline{x}_{t-1}$ and $\overline{x}_{t-1}$ are upper and lower bounds defining the hyperrectangle of states to propagate. The environment $\mathbf{u}_t$ is broken into the static vectors P and x_f and the set X_0 defined in Eq. (8.3), since path planning with the RNN occurs over static environments to static final conditions from a set of possible initial conditions. Under the definitions of $\Phi(\cdot)$, X_{t-1}, and $\mathbf{u}_t$, a method must be devised to approximate bounds $\underline{x}_t$ and $\overline{x}_t$ for the output set X_t defined in Eq. (8.4).

$$X_{t-1} = \{x \in \mathbb{R}^N \mid \underline{x}_{t-1} \leq x \leq \overline{x}_{t-1}\} \tag{8.2}$$

$$X_0 = \{x \in \mathbb{R}^N \mid \underline{x}_0 \leq x \leq \overline{x}_0\} \tag{8.3}$$

$$X_t = \{x \in \mathbb{R}^N \mid \underline{x}_t \leq x \leq \overline{x}_t\} \tag{8.4}$$

To determine the bounds $\underline{x}_t$ and $\overline{x}_t$, [58] proposes a layer-wise propagation formulated from a proof centered on the single layer expression of $\zeta = \phi(W\eta + \theta)$, where ϕ is the activation function, W is the weight matrix, θ is the bias matrix, and η and ζ are the input and output vectors. Provided an input set H bounded by upper and lower bounds $\overline{\eta}_j$ and $\underline{\eta}_j$ for $j \in \{1,...,n_\eta\}$, where n_η is the size of the input vector η, the upper and lower bounds of the propagated set for the output vector Z, composed of the upper and lower bounds $\overline{\zeta}_i$ and $\underline{\zeta}_i$ for $i \in \{1,...,n_\zeta\}$ with n_ζ as the size of the output vector, can be determined by

$$\underline{\zeta}_i = \min_{\eta \in H} \phi\left(\sum_{j=1}^{n_\eta} \omega_{i,j}\eta_j + \theta_i\right) \tag{8.5}$$

$$\overline{\zeta}_i = \max_{\eta \in H} \phi\left(\sum_{j=1}^{n_\eta} \omega_{i,j}\eta_j + \theta_i\right). \tag{8.6}$$

As pointed out in [58], the use of monotonic activation functions (common in NNs) results in Eq. (8.5) and Eq. (8.6) being satisfied by the η solutions to

$$\underline{\zeta}_i = \phi\left(\min_{\eta\in H}\sum_{j=1}^{n_\eta}\omega_{i,j}\eta_j + \theta_i\right) \tag{8.7}$$

$$\overline{\zeta}_i = \phi\left(\max_{\eta\in H}\sum_{j=1}^{n_\eta}\omega_{i,j}\eta_j + \theta_i\right). \tag{8.8}$$

To further simplify the problem, the min/max problems proposed in Eq. (8.7) and Eq. (8.8) are solved as

$$\underline{\zeta}_i = \phi\left(\sum_{j=1}^{n_\eta}\underline{g}_{i,j}\right) \tag{8.9}$$

$$\overline{\zeta}_i = \phi\left(\sum_{j=1}^{n_\eta}\overline{g}_{i,j}\right), \tag{8.10}$$

where $\underline{g}_{i,j}$ and $\overline{g}_{i,j}$ are defined as

$$\underline{g}_{i,j} = \begin{cases} \omega_{i,j}\underline{\eta}_j + \theta_i, & \omega_{i,j} \geq 0 \\ \omega_{i,j}\overline{\eta}_j + \theta_i, & \omega_{i,j} < 0 \end{cases} \tag{8.11}$$

$$\overline{g}_{i,j} = \begin{cases} \omega_{i,j}\overline{\eta}_j + \theta_i, & \omega_{i,j} \geq 0 \\ \omega_{i,j}\underline{\eta}_j + \theta_i, & \omega_{i,j} < 0. \end{cases} \tag{8.12}$$

For a given layer, the bounding set Z is defined by the resulting values of the upper and lower bounds $\overline{\zeta}_i$ and $\underline{\zeta}_i$ for $i \in \{1,...,n_\zeta\}$ where n_ζ is the size of the output vector. This bounding set is then formulated as the input bounding set to the next layer of the network and the process continues until the output bounding set of the last layer is found.

The larger the difference is between upper and lower bounds on an input set H, the larger the observed difference is in the propagated set. To combat this issue, [58] proposes the use of segmented sets to propagate larger regions. These regions result in upper and

lower bound pairs that are adjacent to other regions, e.g. $\overline{\eta}_1$ of set 1 serves as $\underline{\eta}_1$ of its adjacent region. Graphically speaking, this corresponds to a grid of regions extended to multiple dimensions. The following definition provides a stronger view of these regions.

Formally (in the context of the NARX RNN structure), provided upper and lower bounds $\overline{x}_0$ and $\underline{x}_0$ for the input set X_0 and upper and lower bounds $\overline{x}_{t-1}$ and $\underline{x}_{t-1}$ for the input set X_{t-1}, the intervals $X_{0,i} = [\underline{x}_{0,i}, \overline{x}_{0,i}]$ for $i \in \{1,...,N\}$ and $X_{t-1,i} = [\underline{x}_{t-1,i}, \overline{x}_{t-1,i}]$ for $i \in \{1,...,N\}$ are each partitioned into M_i segments. These are defined as $X_{0,i,1} = [x_{0,i,0}, x_{0,i,1}]$, $X_{0,i,2} = [x_{0,i,1}, x_{0,i,2}]$, . . . , $X_{0,i,M} = [x_{0,i,m-1}, x_{0,i,m}]$, where $x_{0,i,m}$ for $m \in \{0,1,...,M_i\}$ are defined as,

$$x_{0,i,m} = \underline{x}_{0,i} + \frac{m\,(\overline{x}_{0,i} - \underline{x}_{0,i})}{M_i}. \tag{8.13}$$

Similarly, $X_{t-1,i,1} = [x_{t-1,i,0}, x_{t-1,i,1}]$, $X_{t-1,i,2} = [x_{t-1,i,1}, x_{t-1,i,2}]$, . . . , $X_{t-1,i,M} = [x_{t-1,i,m-1}, x_{t-1,i,m}]$, where $x_{t-1,i,m}$ for $m \in \{0,1,...,M_i\}$ are defined as,

$$x_{t-1,i,m} = \underline{x}_{t-1,i} + \frac{m\,(\overline{x}_{t-1,i} - \underline{x}_{t-1,i})}{M_i}. \tag{8.14}$$

The segmented regions are then constructed as

$$\mathscr{P}_i = X_{0,1,m_1} \times ... \times X_{0,N,m_n} \times X_{t-1,1,m_{n+1}} \times ... \times X_{t-1,N,m_{n+n}},\ \ i \in \{1,2,...,\prod_{s=1}^{N} M_s \prod_{s=1}^{N} M_s\} \tag{8.15}$$

where $\{m_1,...,m_n,m_{n+1},...m_{n+n}\} \in \{1,...,M_1\} \times ... \times \{1,...,M_N\} \times \{1,...,M_1\} \times ... \times$ - $\{1,...,M_N\}$.

Application of this segmentation approach to propagate an initial state boundary from the Problem Scenario 1 application in Chapter 5, under high segmentation counts of $M_i = 50$ for all state dimensions, was applied with $\underline{x}_0 = [-5.5,\ \ 6.0,\ \ -0.001,\ \ -0.001]$,

$\bar{x}_0 = [-4.5,\ 7.0,\ 0.001,\ 0.001]$, $\underline{x}_{t=0} = [-5.5,\ 6.0,\ -0.1,\ -0.1]$, and $\bar{x}_{t=0} = [-4.5,\ 7.0,\ 0.1,\ 0.1]$. Generation of the bounded sets took on the order of 300 seconds. The results, shown in Fig. 8.1, display a rapid explosion in the bounded sets (shown as the blue boxes) over the course of the RNN CL path, while sample paths generated from initial conditions contained within the initial bounding sets do not exhibit any exploding behaviors. The discrepancy between the bounding set and that of the sampled paths provides motivation for the following revisions created for this thesis.

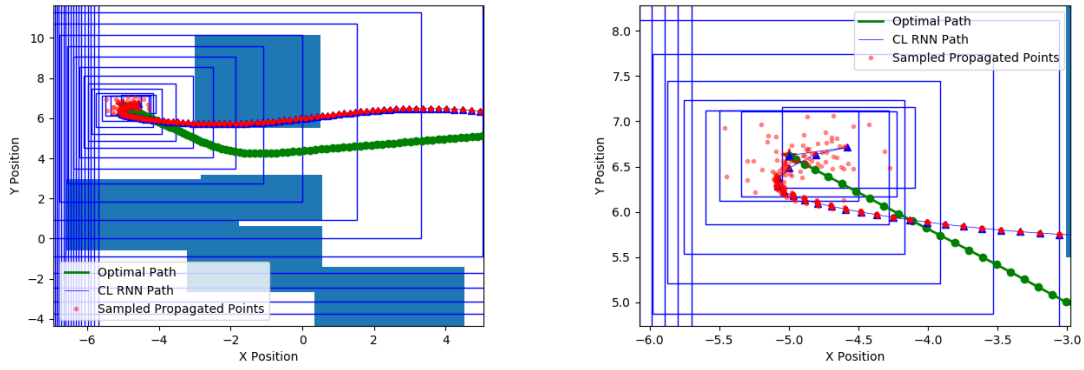

Figure 8.1: The left image (overview) and right image (close up of initial propagation steps) showcase the use of the bounded set propagation presented in [58] with M_i divisions equal to 50 on a whole-path trained RNN from Problem Scenario 1. The blue rectangle outlines represent sequential bounded set propagation, while the red paths are sampled initial conditions from the initial bounded set propagated through the RNN. As shown in both images, the sets explode quickly through time, showcasing the compounding over approximations. Total propagation time took approximately 300 seconds.

Generation of an algorithm to increase the accuracy of the bounded set propagation and speed of execution began with two primary assumptions. First, provided the segmented regions of $\mathscr{P}$ constituting a bounding set input for propagation, the bounds of the output region X_t constructed from the assembly of each $\mathscr{P}_i$ propagation are formed solely

by the edge regions in $\mathscr{P}$. Edge regions of the initial input set are any region that contain bound values of the ultimate bounding values of $\overline{x}_0$, $\underline{x}_0$, $\overline{x}_{t-1}$, or $\underline{x}_{t-1}$. Second, under M_i segment numbers, the edge regions that correspond to resulting upper and lower bounds $\overline{x}_t$ and $\underline{x}_t$ of the propagated output region contain the regions that correspond to the upper and lower output bounds for higher M_i segment numbers. The first assumption implies that as M_i approaches infinity, the output boundary values correspond to singular points along the edge of the initial bounded set sent through the NN. The second assumption implies that these points can be isolated by starting with rough segmentation of the input set, isolating the edge regions that correspond to the boundary values on the output set, and further segmenting those regions and finding the regions within those that correspond to boundary values on the output set. Repeated execution quickly converges to the desired input points that produce the boundaries on the output set. This algorithm is outlined in Algorithm 4.

Algorithm 4: Modified Bounded Set Propagation

procedure COMPUTE OUTPUT BOUNDED SET $X_t(\Phi, X_0, X_{t-1}, M_i, R)$ ▷ Propagate current bounded state set X_{t-1} forward in time to step t
    Compute $\mathscr{P}$ from X_0, X_{t-1}, and M_i
    $\mathscr{H} \leftarrow Edges(\mathscr{P})$
    Propagate regions of $\mathscr{H}$ forward
    Determine $\overline{x}_{t,i}$ and $\underline{x}_{t,i}$ from propagated $\mathscr{H}$
    Determine $\mathscr{H}_i \in \mathscr{H}$ that produced $\overline{x}_{t,i}$ and $\underline{x}_{t,i}$
    for j in $\{1,...,R\}$ **do**
        Segment all $\mathscr{H}_i$ into M_i subregions, creating subregion set $\mathscr{H}_{sub}$
        Propagate regions of $\mathscr{H}_{sub}$ forward
        Determine $\overline{x}_{t,i}$ and $\underline{x}_{t,i}$ from propagated $\mathscr{H}_{sub}$
        Determine $\mathscr{H}_i \in \mathscr{H}_{sub}$ that produced $\overline{x}_{t,i}$ and $\underline{x}_{t,i}$

This algorithm was tested on the same scenario presented in Fig. 8.1 with $M_i = 7$ and $R = 20$, and the results produced are shown in Fig. 8.2. As shown, the bounded sets

do not explode, and all sampled paths are contained in the bounded sets. Additionally, only 130 seconds were required to compute these propagated sets in total. Repeated experiments produced the same resulting behavior.

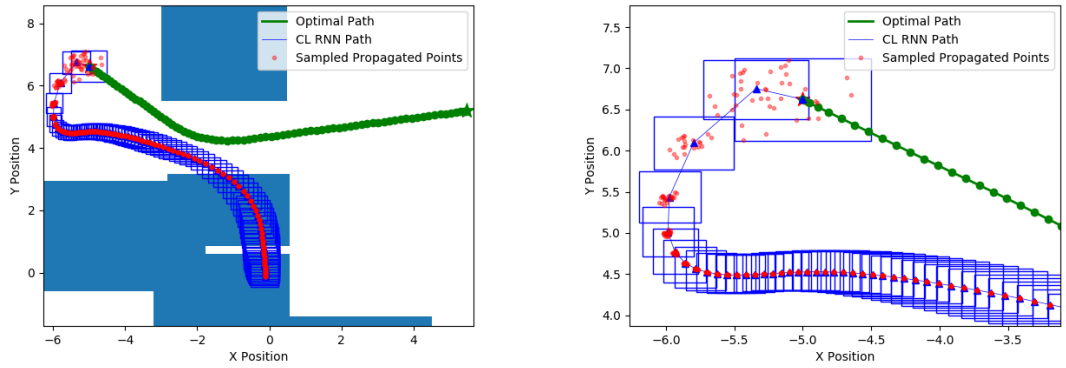

Figure 8.2: The left image (overview) and right image (close up of initial propagation steps) showcase the use of the modified bounded set propagation presented in Algorithm 4 ($M_i = 7$ and $R = 20$) on a poorly whole-path trained RNN from Problem Scenario 1. The blue rectangle outlines represent sequential bounded set propagation, while the red paths are sampled initial conditions from the initial bounded set propagated through the RNN. As shown in both images, sets maintain much tighter approximations even on a poorly trained result while containing all sampled paths as desired. Computation time took approximately 130 seconds.

Experimental results alone do not prove that the algorithm is correct but instead imply that a proof may exist, and further work is required to find such. A formal proof would involve validating the two assumptions that edge segments in the input set correspond to the bounding output segments and that such edge segments also contain the bounding input segments for larger values of M_i. If proven correct, this bounded set algorithm will not only cut down on the time required to compute the reachable sets of a path planning RNN, but it will also provide much more accurate results. Reachability analysis through these bounded set propagations will help enable RNN verification over sets of initial conditions

for static environments, ensuring that no constraints are violated by the RNN. Future work should also investigate utilizing the structure of the propagated bounded sets in informing whole-path network training to better avoid constraint violations.

Chapter 9: Conclusions

This thesis provides contributions in the form of: expanding a kinodynamic RRT* to use Chebyshev collocation optimization; creating a whole-path reinforcement training scheme for RNNs learning state path solutions; making improvements to a bounded set propagation algorithm to improve accuracy and speed of computation; and creating a methodology which uses data generated from optimal kinematic/dynamic path planning solutions to train an RNN to reproduce state paths for use in a path-tracking controller. The end result of the methodology is an RNN path planner that can generate sub-optimal solutions orders of magnitudes faster than the optimization method used to generate them. As it stands, uses of this methodology must consider trade-offs between the speed of solutions obtained and quality of solutions. Furthermore, the improvements to the bounded set propagation algorithm enabled the feasible use of such in the problem scenarios examined, providing a crucial step towards building a formal methods framework for automating the verification of the RNN outputs with respect to desired constraints.

Future work involves a few avenues. First, the proof of the bounded set algorithm must be finished to validate its use. Experimental results indicate that a proof exists, but do not guarantee such. Second, improvements to the whole-path reinforcement training scheme are required to improve the methodology's performance as a whole. As it cur-

rently stands, the whole-path scheme greatly improves closed-loop outputs with respect to training data, but not validation data. Incorporation of policy optimization schemes, such as those shown in [59] and [60], may provide the avenues for better training the network to minimize its error on the desired path over the whole path, helping to better improve learning of the entirety of the path structure. Last, the methodology may be improved by examining the use of the RNN in generating initial guesses for the kinematic/dynamic path planner used. This can speed up the generation of optimal solutions, which then can provide training data faster to the RNN. This feedback loop would create another reinforcement learning layer in which the RNN could be trained to provide solutions that minimized the time that the robust, slower path planner took to find solutions.

Bibliography

[1] Farah Kamil and Khaksar W Zulkifli N. A review on motion planning and obstacle avoidance approaches in dynamic environments. Advances in Robotics & Automation, 04, 01 2015.

[2] A. Valero-Gomez, J. V. Gomez, S. Garrido, and L. Moreno. The path to efficiency: Fast marching method for safer, more efficient mobile robot trajectories. IEEE Robotics Automation Magazine, 20(4):111–120, Dec 2013.

[3] D. Connell and H. M. La. Dynamic path planning and replanning for mobile robots using rrt. In 2017 IEEE International Conference on Systems, Man, and Cybernetics (SMC), pages 1429–1434, Oct 2017.

[4] Dina Youakim and Pere Ridao. Motion planning survey for autonomous mobile manipulators underwater manipulator case study. Robotics and Autonomous Systems, 107:20 – 44, 2018.

[5] M. Kelly. An introduction to trajectory optimization: How to do your own direct collocation. SIAM Review, 59(4):849–904, 2017.

[6] Changliu Liu and Masayoshi Tomizuka. Real time trajectory optimization for nonlinear robotic systems: Relaxation and convexification. Systems & Control Letters, 108:56 – 63, 2017.

[7] M.G. Mohanan and Ambuja Salgoankar. A survey of robotic motion planning in dynamic environments. Robotics and Autonomous Systems, 100:171 – 185, 2018.

[8] Nikolay Jetchev and Marc Toussaint. Fast motion planning from experience: trajectory prediction for speeding up movement generation. Autonomous Robots, 34(1):111–127, Jan 2013.

[9] Hong mei Zhang and Ming long Li. Rapid path planning algorithm for mobile robot in dynamic environment. Advances in Mechanical Engineering, 9(12):1687814017747400, 2017.

[10] Elena Tutubalina, Zulfat Miftahutdinov, Sergey Nikolenko, and Valentin Malykh. Sequence Learning with RNNs for Medical Concept Normalization in User-Generated Texts. arXiv e-prints, page arXiv:1811.11523, Nov 2018.

[11] Zheng Liu and Clair J. Sullivan. Prediction of weather induced background radiation fluctuation with recurrent neural networks. Radiation Physics and Chemistry, 155:275 – 280, 2019. IRRMA-10.

[12] Filippo Maria Bianchi, Enrico Maiorino, Michael C. Kampffmeyer, Antonello Rizzi, and Robert Jenssen. An overview and comparative analysis of recurrent neural networks for short term load forecasting. CoRR, abs/1705.04378, 2017.

[13] Samira Pouyanfar, Saad Sadiq, Yilin Yan, Haiman Tian, Yudong Tao, Maria Presa Reyes, Mei-Ling Shyu, Shu-Ching Chen, and S. S. Iyengar. A survey on deep learning: Algorithms, techniques, and applications. ACM Comput. Surv., 51(5):92:1–92:36, September 2018.

[14] Xiaowei Huang, Marta Kwiatkowska, Sen Wang, and Min Wu. Safety verification of deep neural networks. CoRR, abs/1610.06940, 2016.

[15] Guy Katz, Clark W. Barrett, David L. Dill, Kyle Julian, and Mykel J. Kochenderfer. Reluplex: An efficient SMT solver for verifying deep neural networks. CoRR, abs/1702.01135, 2017.

[16] Alessio Lomuscio and Lalit Maganti. An approach to reachability analysis for feed-forward relu neural networks. CoRR, abs/1706.07351, 2017.

[17] Mounir Ben Nasr and Mohamed Chtourou. Neural network control of nonlinear dynamic systems using hybrid algorithm. Applied Soft Computing, 24:423 – 431, 2014.

[18] S. L. Brunton and B. R. Noack. Closed-Loop Turbulence Control: Progress and Challenges. Applied Mechanics Reviews, 67(5):050801, August 2015.

[19] H.P. Singh and N. Sukavanam. Simulation and stability analysis of neural network based control scheme for switched linear systems. ISA Transactions, 51(1):105 – 110, 2012.

[20] Olalekan P. Ogunmolu, Xuejun Gu, Steve B. Jiang, and Nicholas R. Gans. Nonlinear systems identification using deep dynamic neural networks. CoRR, abs/1610.01439, 2016.

[21] E. De la Rosa, W. Yu, and X. Li. Nonlinear system modeling with deep neural networks and autoencoders algorithm. In 2016 IEEE International Conference on Systems, Man, and Cybernetics (SMC), pages 002157–002162, Oct 2016.

[22] Francesco Leofante, Nina Narodytska, Luca Pulina, and Armando Tacchella. Automated verification of neural networks: Advances, challenges and perspectives. CoRR, abs/1805.09938, 2018.

[23] Ulrich Roth, Marc Walker, Arne Hilmann, and Heinrich Klar. Dynamic path planning with spiking neural networks. In José Mira, Roberto Moreno-Díaz, and Joan Cabestany, editors, Biological and Artificial Computation: From Neuroscience to Technology, pages 1355–1363, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

[24] Simon X Yang and Max Meng. An efficient neural network approach to dynamic robot motion planning. Neural Networks, 13(2):143 – 148, 2000.

[25] Y. Chen and W. Chiu. Optimal robot path planning system by using a neural network-based approach. In 2015 International Automatic Control Conference (CACS), pages 85–90, Nov 2015.

[26] Youssef Bassil. Neural network model for path-planning of robotic rover systems. CoRR, abs/1204.0183, 2012.

[27] Mukesh Kumar Singh and Dayal Parhi. Intelligent neuro-controller for navigation of mobile robot. Proceedings of the International Conference on Advances in Computing, Communication and Control, ICAC3'09, 01 2009.

[28] M. Al-Sagban and R. Dhaouadi. Neural-based navigation of a differential-drive mobile robot. In 2012 12th International Conference on Control Automation Robotics Vision (ICARCV), pages 353–358, Dec 2012.

[29] Xiaoyun Lei, Zhian Zhang, and Peifang Dong. Dynamic path planning of unknown environment based on deep reinforcement learning. J. Robotics, 2018:5781591:1–5781591:10, 2018.

[30] Fabian Schilling, Julien Lecoeur, Fabrizio Schiano, and Dario Floreano. Learning vision-based cohesive flight in drone swarms. CoRR, abs/1809.00543, 2018.

[31] Ahmed H. Qureshi, Mayur J. Bency, and Michael C. Yip. Motion planning networks. CoRR, abs/1806.05767, 2018.

[32] Sertac Karaman and Emilio Frazzoli. Sampling-based algorithms for optimal motion planning. CoRR, abs/1105.1186, 2011.

[33] Dustin J. Webb and Jur van den Berg. Kinodynamic rrt*: Optimal motion planning for systems with linear differential constraints. CoRR, abs/1205.5088, 2012.

[34] S. Stoneman and R. Lampariello. Embedding nonlinear optimization in rrt* for optimal kinodynamic planning. In 53rd IEEE Conference on Decision and Control, pages 3737–3744, Dec 2014.

[35] Jrgen Schmidhuber. Deep learning in neural networks: An overview. Neural Networks, 61:85 – 117, 2015.

[36] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Neurocomputing: Foundations of research. chapter Learning Representations by Back-propagating Errors, pages 696–699. MIT Press, Cambridge, MA, USA, 1988.

[37] P. J. Werbos. Backpropagation through time: what it does and how to do it. Proceedings of the IEEE, 78(10):1550–1560, Oct 1990.

[38] Sebastian Ruder. An overview of gradient descent optimization algorithms. CoRR, abs/1609.04747, 2016.

[39] Léon Bottou. Large-scale machine learning with stochastic gradient descent. In Yves Lechevallier and Gilbert Saporta, editors, Proceedings of COMPSTAT'2010, pages 177–186, Heidelberg, 2010. Physica-Verlag HD.

[40] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. CoRR, abs/1412.6980, 2015.

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

[42] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing Atari with Deep Reinforcement Learning. arXiv e-prints, page arXiv:1312.5602, Dec 2013.

[43] Sen Wang, Daoyuan Jia, and Xinshuo Weng. Deep Reinforcement Learning for Autonomous Driving. arXiv e-prints, page arXiv:1811.11329, Nov 2018.

[44] Marcin Szuster and Zenon Hendzel. Reinforcement Learning in the Control of Nonlinear Continuous Systems, pages 255–297. Springer International Publishing, Cham, 2018.

[45] José Bacelar Almeida, Maria João Frade, Jorge Sousa Pinto, and Simão Melo de Sousa. An Overview of Formal Methods Tools andTechniques, pages 15–44. Springer London, London, 2011.

[46] Luca Bortolussi and Guido Sanguinetti. A statistical approach for computing reachability of non-linear and stochastic dynamical systems. In Gethin Norman and William Sanders, editors, Quantitative Evaluation of Systems, pages 41–56, Cham, 2014. Springer International Publishing.

[47] Hartmut Prautzsch, Wolfgang Boehm, and Marco Paluszny. Bezier and B-Spline Techniques. Springer-Verlag, Berlin, Heidelberg, 2002.

[48] Wei-wei Cai, Le-ping Yang, and Yan-wei Zhu. Bang-bang optimal control for differentially flat systems using mapped pseudospectral method and analytic homotopic approach. Optimal Control Applications and Methods, 37(6):1217–1235.

[49] Fariba Fahroo and I. Michael Ross. Direct trajectory optimization by a chebyshev pseudospectral method. Journal of Guidance, Control, and Dynamics, 25(1):160–166, Jan 2002.

[50] Daniel R. Herber. Basic implementation of multiple-interval pseudospectral methods to solve optimal control problems. Technical report, UIUC-ESDL-2015-01, June 2015.

[51] Philip E. Gill, Walter Murray, and Michael A. Saunders. Snopt: An sqp algorithm for large-scale constrained optimization. SIAM Rev., 47(1):99–131, January 2005.

[52] François Chollet et al. Keras. `https://github.com/fchollet/keras`, 2015.

[53] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[54] Timothy Dozat. Incorporating nesterov momentum into adam. 2015.

[55] Salah Rifai, Pascal Vincent, Xavier Muller, Xavier Glorot, and Yoshua Bengio. Contractive auto-encoders: Explicit invariance during feature extraction. In ICML, 2011.

[56] Jakob Löber. Optimal trajectory tracking. arXiv e-prints, page arXiv:1601.03249, Dec 2015.

[57] Giuseppe Fedele, Luigi D'Alfonso, Francesco Chiaravalloti, and Gaetano D'Aquila. Obstacles avoidance based on switching potential functions. Journal of Intelligent & Robotic Systems, 90(3):387–405, Jun 2018.

[58] Weiming Xiang and Taylor T. Johnson. Reachability analysis and safety verification for neural network control systems. CoRR, abs/1805.09944, 2018.

[59] Ruiyi Zhang, Changyou Chen, Chunyuan Li, and Lawrence Carin. Policy optimization as Wasserstein gradient flows. In Jennifer Dy and Andreas Krause, editors, Proceedings of the 35th International Conference on Machine Learning, volume 80 of Proceedings of Machine Learning Research, pages 5737–5746, Stockholmsmssan, Stockholm Sweden, 10–15 Jul 2018. PMLR.

[60] Timothy P. Lillicrap, Jonathan J. Hunt, Alexand er Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. arXiv e-prints, page arXiv:1509.02971, Sep 2015.