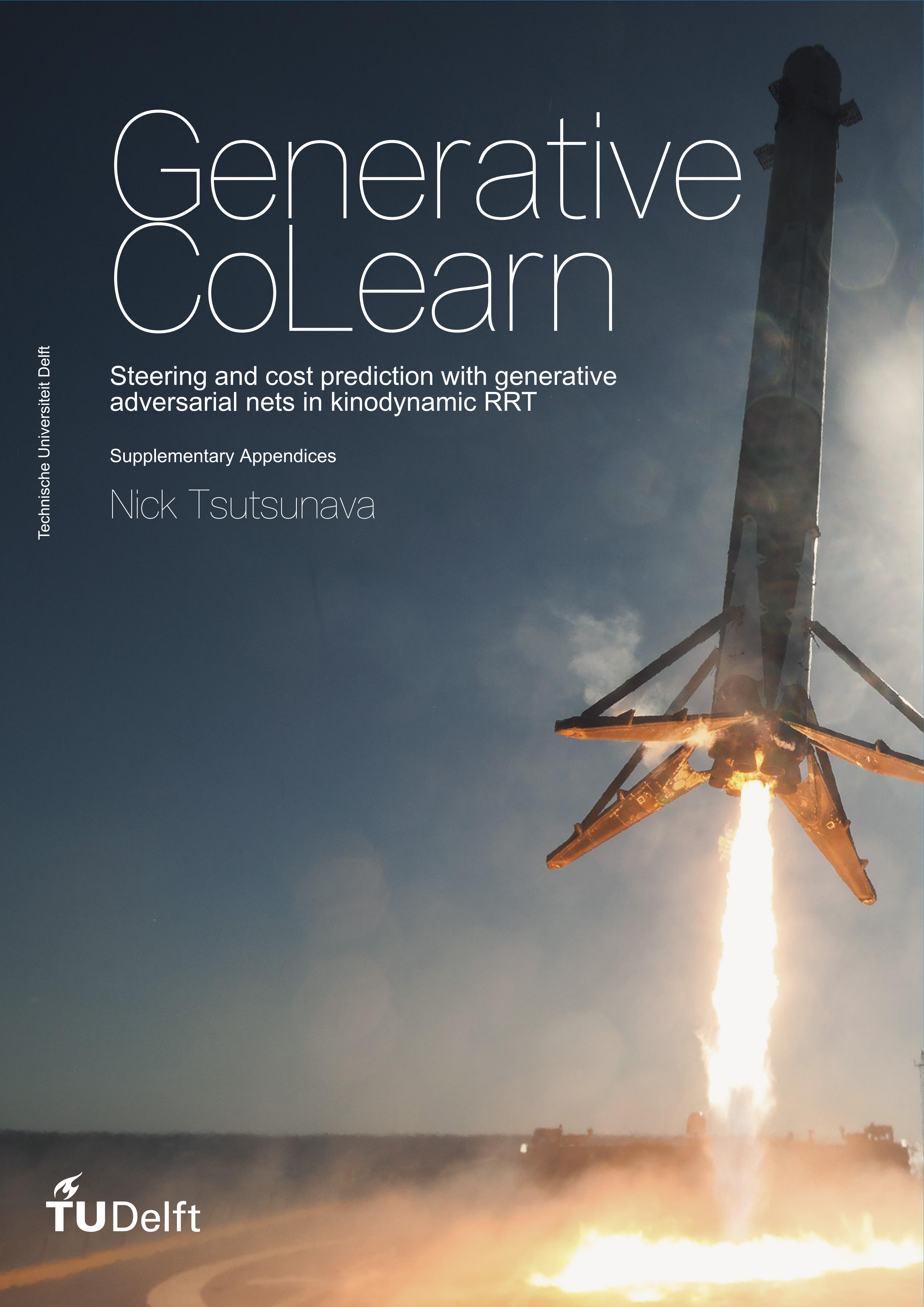


Generative CoLearn

Steering and cost prediction with generative adversarial nets in kinodynamic RRT

Supplementary Appendices

Nick Tsutsunava

Technische Universiteit Delft

Generative CoLearn

Steering and cost prediction with generative adversarial nets in kinodynamic RRT

by

Nick Tsutsunava

to obtain the degree of Master of Science

at the Delft University of Technology,

to be defended publicly on Friday October 5, 2018 at 14:00.

Student number:	4085396	
Project duration:	November 15, 2017 – September 15, 2018	
Thesis committee:	Prof. dr. ir. M. Wisse,	TU Delft, supervisor
	Dr. ing. J. Kober,	TU Delft
	Ir. T.D. de Bruin,	TU Delft
	Ir. W.J. Wolfslag,	TU Delft

Preface

This document is supplementary to the paper *Generative CoLearn: steering and cost prediction with generative adversarial nets in kinodynamic RRT*, for submission to the International Conference on Robotics and Automation 2019. The paper has been written in collaboration with W.J. Wolfslag, a postdoctoral researcher at the University of Edinburgh, United Kingdom. Wolfslag was primarily concerned with the data generation aspect of the research, focussing on the equations of motion and indirect optimal control. My personal contribution involved the complete implementation of the required software, mainly focussing on the machine learning aspect. This work consists of implementation, tuning and benchmarking of the neural nets, implementation of the path planning algorithm and all the necessary analysis. A summary of the system is presented in Appendix B.

Generative CoLearn is an extension to RRT-CoLearn, currently the state-of-the-art kinodynamic planner by Wolfslag et al. [1]. The purpose of this document is to give a detailed overview of the MSc thesis project, especially on the work that has not been covered in the paper. Within these appendices, the reader will find various supplementary figures, which further back up the conclusions in the paper, a thorough breakdown of the various research paths taken during the project and details on implementation. Initially, the architecture of the software stack will be presented to give the reader an understanding of the working parts. Various efforts in terms of machine learning will be presented as well, detailing the reasoning behind the use of Generative Adversarial Networks (GAN). Finally, the implementation of the Rapidly-exploring Random Trees (RRT) will be covered including the effect of several key improvements to the algorithm.

I would like to thank my supervisors, whom are co-authors on the paper, Dr. ir. C. Hernández Corbato, Dr. ir. M. Bharatheesha and Prof. dr. ir. M. Wisse for their insightful comments and feedback. Additionally, I would like to thank my family and friends for their endless support during my Master thesis project.

Nick Tsutsunava
Delft, September 2018

Contents

A Preliminaries

A.1. Introduction

The goal of kinodynamic planning is to plan a path in state space, with kinematic and dynamical constraints. This is computationally difficult since it requires solving a *two-point boundary value problem*, known to be NP-HARD [2]. Therefore, with the current solutions, it is prohibitive to implement kinodynamic planning in real-time applications such as motion planning in autonomous vehicles [3]. Throughout literature, various methods are explored to speed up computation, most notably by means of machine learning and sampling based methods such as Rapidly-exploring Random Trees (RRT). These type of Learning-RRT approaches use an online-offline paradigm where long running tasks, e.g. data generation and training the machine learning algorithm, are shifted to an offline process. As a result, the trained model can then be used to make quick predictions during online planning. With this predictive approach, it is no longer necessary to solve computationally hard problems during path planning. Currently, the state-of-the-art of such systems is RRT-CoLearn, a method to predict both the cost and steering inputs to steer a dynamical system from one state to another [1]. However, its limitation lies in the fact that it uses a simple machine learning algorithm that is inadequate for scaling to systems with more degrees of freedom. As a solution to the limitation of RRT-CoLearn, Generative CoLearn is presented.

The novelty of Generative CoLearn lies in the use of generative adversarial networks (GAN) as the learning algorithm for online predictions during RRT [4]. The GAN is a very popular, deep generative model (DGM) and has seen applications in medical data analysis, robotics and autonomous driving [5–7]. It is capable of learning extremely non-linear relationships from data (i.e. images) after which new and similar data can be generated.

With Generative CoLearn we plan a path in state space for a pendulum and a planar arm using RRT. For each system, we generate short time optimal trajectories with corresponding cost and steering inputs. We train a GAN on this dataset to be able to predict the cost and steering inputs based on a query trajectory. With Generative CoLearn, we see a considerable improvement in performance compared to RRT-CoLearn, attributed to the learning capacity of the GAN. Consequently, we demonstrate path planning on a planar arm.

In these appendices the implementation of Generative CoLearn is detailed. Additionally, supplementary figures and analyses to the conference paper are presented. Appendix B presents the architecture of the software, giving an overview of the working parts. The data generation and subsequent analysis is detailed in Appendix C. In Appendix D the machine learning aspect of Generative CoLearn is discussed, including implementation and evaluation. Finally, the path planning implementation and supplementary results are presented in Appendix E.

A.2. Text Formatting

Different text formatting is used to exemplify specific meaning. Below is a list of the formats used.

- `Typewriter` is used to exemplify code such as classes, filenames, extensions and general components.
- *Italics* is used to indicate importance of a term.

A.3. Acronyms

CGAN: Conditional Generative Adversarial Network

CVAE: Conditional Variational Autoencoder

DGM: Deep Generative Model

GAN: Generative Adversarial Network

KNN: k-Nearest Neighbour

LSGAN: Least Squares Generative Adversarial Network

MSE: Mean Squared Error

RRT: Rapidly-exploring Random Trees

ReLU: Rectified Linear Unit

VAE: Variational Autoencoder

B Architecture

Generative CoLearn is a novel approach to kinodynamic planning. It is mainly written in Python and uses several open-source libraries for its implementation. Even though Generative CoLearn is based on RRT-CoLearn, it is completely written from the ground up. This section will detail the architecture of the system, and the reasoning why it has been rewritten.

At the start of the project, the code for RRT-CoLearn was supplied. However, two main issues necessitated writing the codebase for Generative CoLearn from scratch. First and foremost, the code for RRT-CoLearn is convoluted and difficult to read. Building on top of such a fragile system would only hamper the research. Second, the data generation script, supplied by Wouter Wolfslag, is not compatible with RRT-CoLearn. Rewriting parts of RRT-CoLearn to conform to the data generated by the script would already result in major changes. Therefore, Generative CoLearn is built from the ground up, resulting in a better understanding of the code, a cleaner implementation and modularity.

A thorough yet simple design was created that could support the changing nature of the research. The system is written in Python 3 with the Object Oriented Programming (OOP) design pattern. Benefits of such a pattern include easy extensibility, encapsulation and readability, all of which were driving factors in the implementation of Generative CoLearn. A somewhat simplified schematic of the design is presented in Figure B.1. Each main block will be extensively covered in the next appendices, with brief summaries following later in this section.

The implementation of Generative CoLearn consists out of various subsystems or blocks. These blocks have dependencies with each other and one can visualise these dependencies as stacking them on one another. Hence the term *stack* is used when referring to the complete implementation of the system. These blocks make it easy to swap them out for other implementations such as different machine learning algorithms or other dynamical systems. In fact, in the early stages of the research it was already known that some blocks would require swapping more than others, which prompted the idea of dynamically loading an implementation of a subsystem without explicitly coding dependencies. With a single configuration file it is easy to select which block to use. This greatly reduces development time and increased flexibility overall, at the cost of spending slightly more time during initial implementation.

As is common for Learning-RRT systems, Generative CoLearn employs an offline-online paradigm and is visualised in Figure B.2. Initially, data is generated and used for training, both happening offline. The data contains the initial and final states (x_0, x_1), the steering inputs u, and the steering cost, which are used for training the GAN. The GAN consists of two neural networks, the generator $G(z|y)$ and the discriminator $D(x)$, which are trained adversarially. The trained model can then be used online to make predictions for cost and steering inputs during path planning. Additionally, the discriminator is used as a classifier for determining the feasibility of query trajectories during path planning. The use of GAN speeds up online path planning as solving boundary value problems is no longer necessary. However,

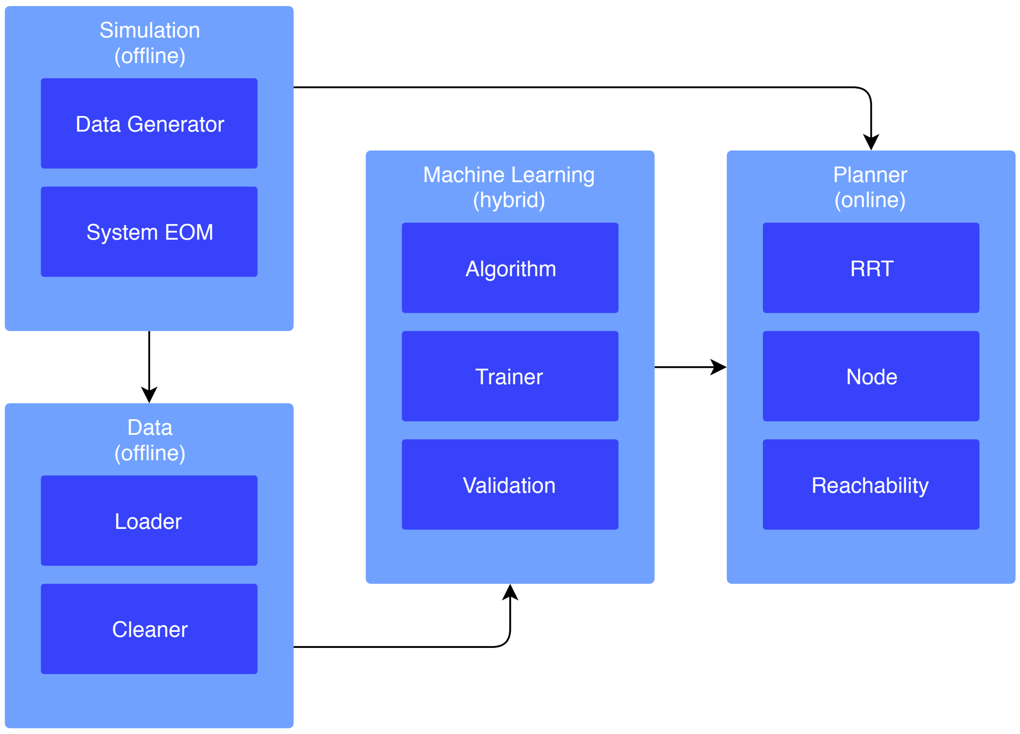


Figure B.1: The implementation of Generative CoLearn is modular. The dependencies are indicated with arrows. For example, the machine learning and simulation components are injected into the planner. Note the difference of what occurs online and offline. Machine learning is a hybrid as training occurs offline and prediction is online.

data generation and training of the learning algorithm do take substantially more time than planning itself, but is considered a reasonable trade-off.

Next, a brief summary will be given of the subsystems. Their detailed implementation and explanation is covered in corresponding appendices. The code for Generative CoLearn is fully open-sourced and is available on GitHub[1].

B.1. Simulation

The `Simulation` subsystem contains the equations of motion of a dynamical system and a method for generating data for the machine learning model. There are two subsystems that depend on `Simulation`, namely `Planner`, which uses it for forward simulation and `Data` providing an API for the rest of the stack.

B.2. Data

Once the simulation is complete and the data has been saved as a `.csv` file, the `Loader` class reads the data into memory and provides a simple API to the rest of the system. This API makes sure that the data is available throughout the entire stack and allows for splitting of the data into a training and test set, pre-processing and batch selection. Additionally, the `Cleaner` class implements the cleaning algorithm mentioned in Appendix C.

[1]`http://github.com/ortix/generative-colearn`

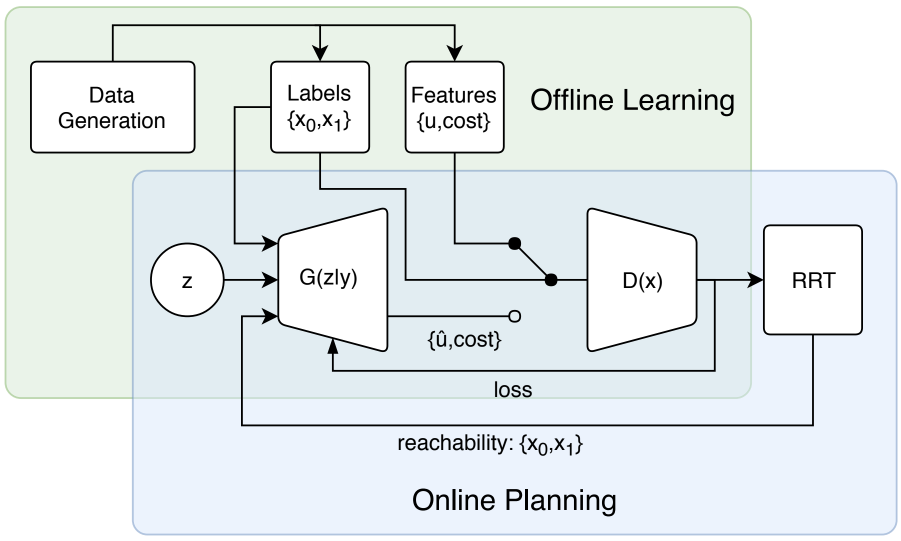


Figure B.2: Generative CoLearn is split up in two phases. First, data is generated and used for training the GAN. Next, the trained model is used to enable fast path generation.

B.3. Machine Learning

At the core of Generative CoLearn lies machine learning. In particular, a deep generative model (DGM), implemented with *neural networks* is used as a function approximator. Such a DGM can be conditioned on auxiliary information (supervised learning) such that it can predict values based on a query. The `Algorithm` is trained on the generated data consisting of trajectories and corresponding steering inputs and cost by the `Trainer`. In this text this is often referred to as a *trained model* or simply *model*. Afterwards, the model is validated with a variety of tests from the `Validation` class. The model is then available throughout the system and has a uniform interface for predicting values.

B.4. Planner

Finally, all the parts come together during path planning using Rapidly-exploring Random Trees (RRT). Both the trained model and the `Simulation` class are injected into `RRT` for prediction and forward simulation. The `Node` class serves as a wrapper for representing the state during path planning with some helpful functions. The `Reachability` class ensures that only nodes that meet a certain criteria are considered for expansion during planning.

C Simulation and Data Generation

In machine learning applications, a large amount of data is necessary in order to successfully train a model. Generative CoLearn is no exception as it relies on many examples of short time optimal trajectories with corresponding steering inputs and cost for training. Since the data generation and simulations are implemented by Wouter Wolfslag, these systems are mostly regarded as black-box during the research. Therefore, the theory will not be covered in depth. Instead, this section details generation of the data and its analysis.

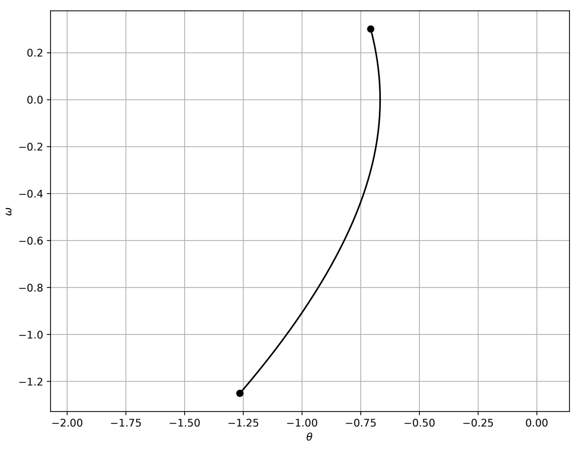
C.1. Time Optimal Trajectories

A challenge in kinodynamic planning is that the steering inputs are difficult to learn. This is caused by the large amount of variables that parametrise the steering input [1]. Similar to RRT-CoLearn, Generative CoLearn aims to solve this by formulating the steering function as an indirect optimal control problem, simplifying and reducing the amount of inputs. Per degree of freedom the steering input can be defined by two parameters, also known as the *costates* λ and μ. The costate μ directly affects the initial torque of the system while the costate λ mainly affects the time until μ switches sign resulting in a bang-bang type control. For trajectories where the sign of μ does not switch, a large range of values for λ are allowed. This occurs where the simulation time is shorter than the time for the switch to occur. In the case that the sign of μ does switch, there is a one-to-one relationship between the costates. Such trajectories are shown in Figure C.1.

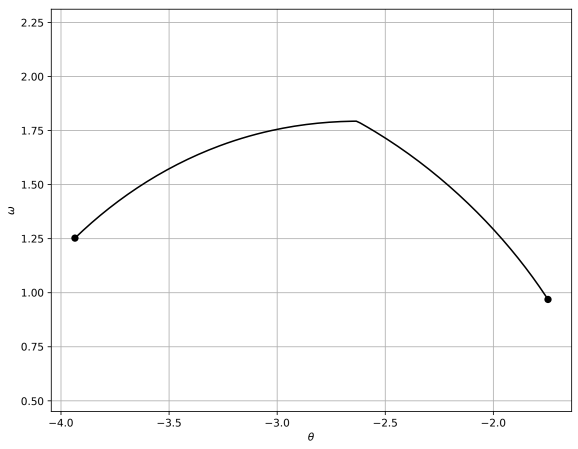
The machine learning approach in kinodynamic RRT is to learn by example. In Generative CoLearn, these examples are short time optimal trajectories of a pendulum and planar arm. The trajectories are generated by a forward simulation with randomly sampled costates and an initial state (θ_0, ω_0), resulting in a final state (θ_1, ω_1). The initial states are uniformly sampled over their domain. For simplification in learning and analysis, the costates are sampled from a unit circle for the pendulum and a 4D-sphere for the planar arm. Therefore, the domain for each costate is constrained to (-1,1), resulting in the norm of the costate vector to be unity. This feature is key in analysis of the learning performance.

C.2. Data Generation

Data generation for the pendulum is realised directly from within Python with the corresponding equations of motion. The pendulum swing-up task involves moving from $(\theta, \omega) = (-\pi, 0)$ to $(0, 0)$. Therefore, initial states are sampled according to



(a) Trajectory without switching torque



(b) Trajectory with switching torque

Figure C.1: If λ is sampled such that the torque does not switch within the simulation time, a trajectory is generated as shown in C.1b. If the simulation time is sufficiently long for μ to switch, a trajectory is generated as shown in C.1a.

Table C.1: The dataset is divided into features and labels for the machine learning algorithm.

	Features				Labels			
Row	λ_0	μ_0	t_f	cost	θ_0	ω_0	θ_1	ω_1
1	0.951	-0.306	0.840	0.840	0.357	-0.575	0.135	0.016
2	0.872	-0.488	0.900	0.900	-4.434	-1.619	-5.297	-0.324

$$\theta \sim \mathcal{U}(-\frac{3}{2}\pi, \pi) \tag{C.1}$$

$$\omega \sim \mathcal{U}(-\pi, \pi). \tag{C.2}$$

For the initial conditions of θ a larger range is used to facilitate enough examples where swinging back- and forth is possible. The link length and mass are set to unity. The maximum torque is $\tau_{\text{max}} = 0.5$. These initial conditions together with the equations of motion result in final states and are stored in a `.csv` file. In total 40000 simulated samples are generated by means of forward simulation using the classical Runge-Kutte method for numeric integration. An example of what is stored in the `.csv` file is shown in Table C.1. The costates along with the cost are the **features** to be learned by the machine learning algorithm. The initial and final states are the **labels** with which the machine learning algorithm is conditioned. Note that the features include an additional variable t_f, which is the simulation time. As the optimal trajectories are *time* optimal, the cost is equal to the simulation time. This feature duplication does not negatively affect training. Additionally, separating cost and simulation time allows for future changes in the type of optimal control, such as *energy* optimal, which has a different cost.

The equations of motion and dataset generation for the planar arm dataset is realised with Julia. In order to make use of the simulation, a Python-Julia interface (`pyjulia`) is necessary, leading to overhead and slower planning times. Due to limitations in the interface, data can not be generated directly from within the Generative CoLearn stack and requires running the generation script manually.

The task for the planar arm involves moving from $(\theta_0, \theta_1, \omega_0, \omega_1) = (-0.25\pi, 0, 0, 0)$ to $(0.25\pi, 0, 0, 0)$. As such, the initial states for the planar arm are sampled according to

$$(\theta_1, \theta_2) \sim \mathcal{U}(-0.5\pi, 0.5\pi) \tag{C.3}$$

$$(\omega_1, \omega_2) \sim \mathcal{U}(-1, 1) \tag{C.4}$$

Table C.2: As the degrees of freedom grow, so does the dimensionality of the dataset. For space saving reasons, values have been omitted.

	Features						Labels							
Row	λ_0	μ_0	λ_1	μ_1	t_f	cost	θ_{00}	θ_{01}	ω_{01}	ω_{02}	θ_{11}	θ_{12}	ω_{11}	ω_{12}
1	-0.895		⋯			0.24	-0.742			⋯				-0.965

Table C.3: The cleaning of the dataset is not necessary as reducing the amount of overlapping trajectories has a detrimental effect on KNN, which is the machine learning algorithm used by RRT-CoLearn.

d	0	0.1	0.2	0.3	0.4	0.5
Error	0.09	0.09	0.12	0.16	0.17	0.21
Nodes	156	166	270	327	447	549
Fail rate	40%	43%	32%	24%	14%	24%

and are used as the parameters for the equations of motion in the forward simulation. The link lengths, masses and torques are all set to unity. In order to increase state-space coverage, half of the total samples are simulated backwards in time. This is accomplished by using the initial states as final states and integrating the equations of motion from $t = t_f$ to $t = 0$. Similar to the pendulum, the classical Runge-Kutte method is used for numeric integration. Generating 500000 samples in Julia takes around 4-6 hours, depending on the machine. As such, the data is only generated once for the experiments, as opposed to the pendulum dataset, which is generated for each epoch. An example of the generated dataset is shown in Table C.2

C.3. Data Cleaning

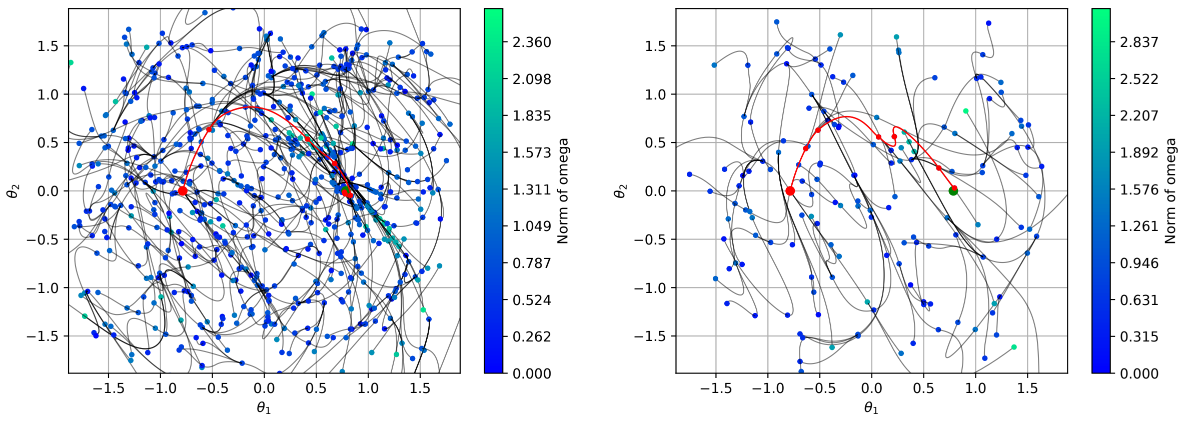
An inherent issue to the data generation method of RRT-CoLearn, results in overlapping trajectories with different costs and costates. When the trained machine learning algorithm is queried with a trajectory during RRT, it looks up several similar trajectories it has seen and averages the corresponding costates. Due to the parametrisation of the costates in RRT-CoLearn, this averaging results in invalid trajectory-costate combinations. A cleaning algorithm solves this issue by reducing the amount of overlapping trajectories, improving the performance of the algorithm.

The original premise of Generative CoLearn was to avoid cleaning by using a more powerful machine learning algorithm that did not average the costates. Even though trajectories in Generative CoLearn also overlap, they do not adversely affect the predictions due to the re-parametrisation of the costates. As a result, cleaning of the dataset is longer necessary. This is experimentally confirmed by implementing and running the cleaning algorithm from RRT-CoLearn. The cleaning algorithm is parametrised by d which is the Euclidean distance between trajectory vectors $(\theta_0, \omega_0, \theta_1, \omega_1)$ in the dataset and can be interpreted as a measure of how much trajectories overlap. Trajectories that lie within d to each other are removed. This process is repeated multiple times until there are no longer trajectories that overlap. After the cleaning process, we run Generative CoLearn with KNN[1] and observe that the planning performance is adversely affected. This is repeated for various values for d of which the results are summarised in Table C.3. From this table we can confirm that cleaning is not necessary and has a detrimental effect on the performance, as it simply reduces the amount of samples to learn from. It is still unknown why exactly the fail rate decreases. However, the fail rate is considered a robustness measure, whereas the amount of nodes and error are true performance measures. As such, the conclusion that cleaning is detrimental to the performance does not change.

[1]More details on this learning algorithm are presented in Appendix D.

C.4. Data Coverage

In order for the learning algorithm to generalise well, the generated data needs to cover the state-space sufficiently. Even though the planar arm does converge with limited coverage, it results poor accuracy near the goal with a relatively high node count. This can clearly be observed in Figure C.2a by two features. First, the clustering of nodes near the goal region is visible. Second, straight lines crossing the goal region are visible, which can be interpreted as the end-effector overshooting. Both features are indicative of the GAN having difficulty predicting costates that result in a sufficiently low angular velocity near the goal state. A reasonable heuristic is to increase the amount of samples in the dataset to increase state space coverage. However, doubling the samples from 250000 to 500000 resulted in only a slight improvement. The question then arises: *how much data do we need?*. In other words, we are interested in finding how the amount of data affects the performance of GAN during planning. This section will detail the analysis of the generated data that led to the augmentation of the dataset, which resulted in a substantial improvement in convergence for the planar arm, shown in Figure C.2b.



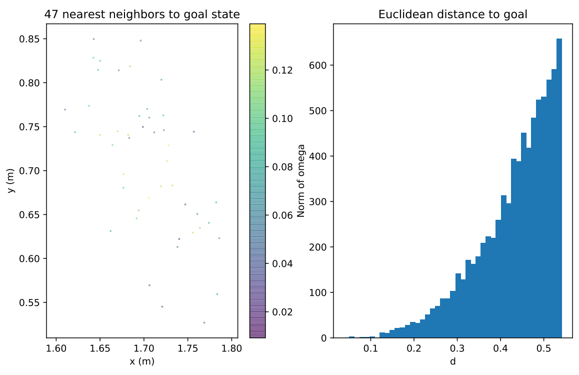
(a) Without data augmentation (b) With data augmentation

Figure C.2: The GAN has difficulty predicting costates resulting in low velocity states near the goal. Augmenting data with more examples of low velocity states, results in improved convergence.

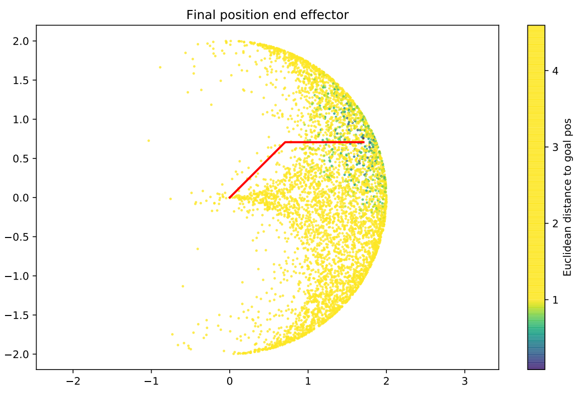
For the RRT algorithm to converge, a Euclidian distance of 0.15 between the final state and goal state is required. This means that the learning algorithm must be able to generate the corresponding costates that result in a final state near the goal state. A look in the generated dataset of 500000 samples reveals that there are merely 26 samples from which the learning algorithm needs to learn. This tiny fraction will almost certainly cause the GAN to miss this mode during training. Unfortunately, doubling the size of the dataset only increased the amount of samples that fall within the 0.15 threshold to 47. In fact, a problem was that the velocities of the final state were too high as these are sampled uniformly during generation. This was especially observed during RRT where the algorithm converged to the position, while the velocity was still far too high.

Figure C.4 gives more insight into the issue by converting 5000 random training samples from state-space to task space. We indeed see that the coverage is extremely limited near the goal region. There are almost no points available below the threshold of 0.15 for the learning algorithm, as shown in Figure C.4a. In Figure C.3b it can be seen that even though the position space is sufficiently covered, the velocities are not within the necessary range, resulting in a high Euclidean distance.

A remedy for the low state-space coverage near the goal is to simply generate more points there. The reasoning behind this approach is to force the learning algorithm to capture a specific mode. This can give us insight in how an increase in state-space coverage affects convergence. Augmenting the dataset in such a way is more efficient than generating an order of magnitude more data points. Consequently, an additional 100000 points are generated near the start and goal regions. This is done by sampling the initial states from a normal distribution instead of a uniform distribution. The angles and velocities for both start and goal positions are sampled as



(a)



(b)

Figure C.3: The sparsity of the state-space coverage is a limiting factor for the learning algorithm to generalise.

$$(\theta_1, \theta_2) \sim (\mathcal{N}(\pm 0.25, 0.1), \mathcal{N}(0, 0.1)) \tag{C.5}$$

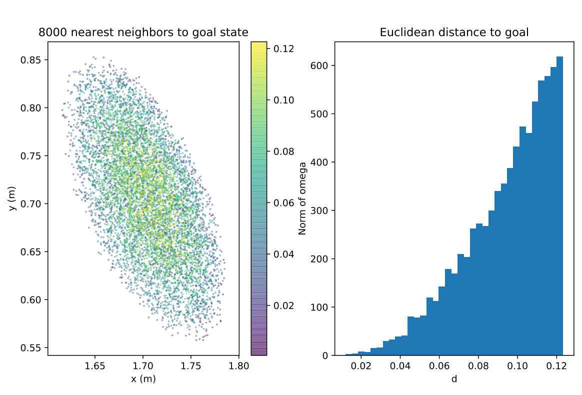
$$(\omega_1, \omega_2) \sim \mathcal{N}(0, 0.1\mathbf{I}_2). \tag{C.6}$$

The dataset generated with this sampling method is merged to the already existing dataset and used for training. This augmented dataset results in an improved coverage near the goal. If we plot the task-space again, we see that now the goal region is sufficiently covered with enough points. As a result of this augmentation, the node count decreased drastically from around 600 to around 100 with improved accuracy near the goal, as shown in Figure C.2b.

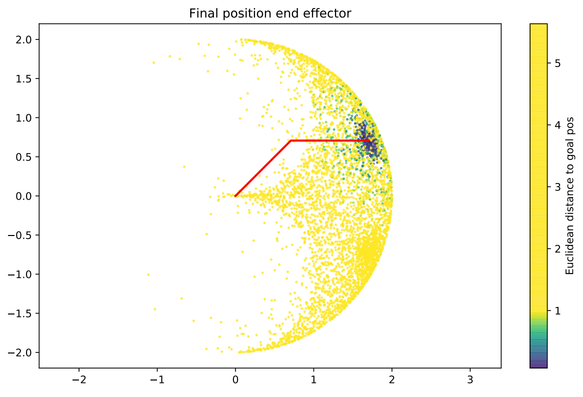
Recall the previously posed question: *how much data do we need?* It is not possible to answer the question with a number, but it has become clear from these results that substantially more data results in improved performance. This comes at the cost of longer generation and training times. Note that this is not necessarily a solution, but rather an observation of how the amount of data affects performance.

There are two potential issues with the GAN that could explain the data requirement[2]. First, the conditioning of the the networks could be too aggressive resulting in poor generalisation. This would lead to the GAN learning one-to-one relationships between trajectories and costates across the entire state-space. Second, the GAN might suffer mode collapse where it simply recreates what it has seen during training. These are hard to debug issues as there is no simple way to observe if the generated samples make sense and contain enough variety. Even though the aforementioned issues with the GAN are hypothetical, reducing the amount of required data is still a standing challenge and left for future work.

[2]It is advised to first read Appendix D before continuing.



(a)



(b)

Figure C.4: Augmenting the dataset by sampling initial states near the goal during generation results in improved state-space coverage.

D Machine Learning

The main contribution of Generative CoLearn is the novel use of a machine learning algorithm. Similar to RRT-CoLearn, machine learning is used in Generative CoLearn to approximate the steering cost and steering inputs during path planning, tremendously speeding up the process. However, the algorithm used in RRT-CoLearn is limited in terms of computation speed and generalisation. Therefore, we aim to use a deep generative model (DGM) for cost and steering predictions. In a preliminary literature survey, it was found that the two most popular deep generative models are the Variational Autoencoder (VAE) and Generative Adversarial Networks (GAN) [8, 9]. Both models are implemented and tested. From analysing various benchmarks, GAN is superior over VAE in the context of Generative CoLearn. This appendix will detail the choice and implementation of these deep generative models. Furthermore, an analysis of the performance of GAN is given and compared to the learning algorithm used in RRT-CoLearn.

D.1. k-Nearest Neighbour

One of the most simple machine learning algorithm is k-nearest neighbour (KNN). It is a *non-parametric* learning algorithm in the sense that all the available data is stored in memory and used for prediction. Generally, KNN is used as a classification algorithm, but is also suitable for regression problems by means of interpolation. The use of KNN is simple. The available training data is stored in a k-dimensional tree. When the algorithm is queried, the k nearest neighbours to the query are looked up in the tree and are averaged. This works well for simple linear problems or when a massive amount of data is available. However, for the latter case, the algorithm becomes extremely slow as lookup time for tree traversal grows with data dimensionality. This is one of the reasons why the KNN algorithm is not suitable for learning based kinodynamic RRT applications where real-time performance is critical.

The implementation of KNN in Generative CoLearn is similar to RRT-CoLearn. The KNN library from *scikit-learn* is used for integration in the stack [10]. Similar to RRT-CoLearn, $k = 3$, the distance metric is Euclidean and the leaf size is set to 10.

In the next section we will investigate other machine learning algorithms, which can alleviate the limitations of KNN. Nevertheless, KNN will be used as as baseline benchmark to compare Generative CoLearn to RRT-CoLearn.

Figure D.1: A GAN trained on celebrities is able to generate high quality images of never before seen faces [11].

D.2. Deep Generative Models

In literature the two most widely used deep generative models are the VAE and GAN and are typically implemented with deep neural networks. Their goal is to model a data distribution such that new but similar samples can be generated. For example, a GAN trained on images of celebrities can generate new, never seen before faces, shown in Figure D.1.

There is a notable difference between the GAN and VAE. The VAE models the data distribution explicitly by inference. This means that it is trying to match the data distribution of the ground truth by means of likelihood maximisation. The GAN on the other hand is more flexible and implicitly models the data distribution. Both modelling approaches have their benefits and drawbacks. The VAE on one hand is easy to train and has good generalisation abilities. Unfortunately, it assumes that the underlying features of the data can be mapped to a standard Gaussian distribution. This causes oversimplification of the data resulting in poor samples [12]. The GAN on the other hand, does not make such assumptions. This allows the GAN to generate superior samples for extremely complex and high dimensional data such as images of faces. This comes at the cost of being notoriously difficult to train [13].

Initially, the VAE was implemented as interpolation in its simple and structured latent space seems more appropriate in the setting of path planning in state space. However, the data distribution proved to be too complex for the VAE to capture. As a result, the GAN was implemented with the assumption that its power to learn the data distribution *implicitly* would yield improved results, which ultimately is the case. Furthermore, the original formulation of both VAE and GAN is unconditional and thus training is unsupervised. However, in the context of Generative CoLearn, the learning algorithm needs to be conditioned on the labels (initial and final states), resulting in supervised learning. Note that conditioning deep generative models is still an active field of research and no consensus exists on how to do so. The next sections will cover the implementation of both models and how they are conditioned. After that, benchmark results are presented, which show the learning performance of VAE and GAN.

D.2.1. VAE Implementation

The first algorithm implemented is the VAE and is done in Keras 2.1 (with Tensorflow backend), a high-level API for creating deep learning models. This library was initially chosen due to its simplicity compared to Tensorflow, with readily available resources for creating the VAE. The implementation of the VAE on the Keras blog[1] was used as a reference design for the actual implementation in the stack. The main components of the model are the encoder, decoder and the sampler for the latent space.

[1]`https://blog.keras.io/building-autoencoders-in-keras.html`

The core idea behind the VAE is to model the dataset with a normal distribution by outputting its parameters, μ and σ^2, from the encoder network. This is achieved by the recognition model $q_\phi(z|x)$, which is a parametrised probability distribution implemented with a neural network. Formally the encoder network represents

$$q_\phi(z|x) = \mathcal{N}(z|\mu(x), \sigma(x)^2) \tag{D.1}$$

The decoder then samples a standard normal distribution, known as the *latent space* according to $z \sim \mathcal{N}(0, \mathbf{I}_2)$. Similar to the encoder, the network also represents a probability distribution and is known as the likelihood $p_\theta(x|z)$. Consequently, the decoder reconstructs the input sample x. This reconstruction represent the mean of a non-linear transformation of the sampled latent space by the decoder. Both the encoder and decoder are jointly trained to balance two losses, formalised with the following equation.

$$\mathcal{L}(x, \theta, \phi) = \underbrace{\mathbb{E}_{q_\phi(z|x)} \log p_\theta(x|z)}_{\text{reconstruction loss}} - \underbrace{D_{KL}\left[q_\phi(z|x)||p_\theta(z|x)\right]}_{\text{latent loss}} \tag{D.2}$$

Equation (D.2) is known as the expectation lower bound (ELBO). The aim is to maximise the expectation that the reconstructed sample originates from the true data distribution $p(x)$, while simultaneously minimising the KL divergence between the recognition model and the latent space. Since maximising the expectation is intractable we instead minimise the ELBO, measured as the mean squared error between input and output [8].

The code snippets for the losses are shown in Figure D.2. The KL loss is solved analytically since we assume that the latent space is a Gaussian distribution as presented in the original paper. Note that for the pendulum dataset it was observed that the KL loss almost immediately converged to zero. Therefore, a manually tuned regulariser term was added, which ultimately did not have any effect.

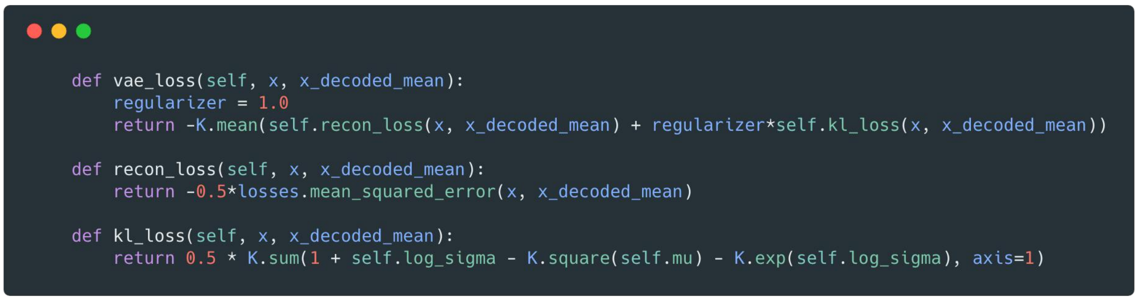
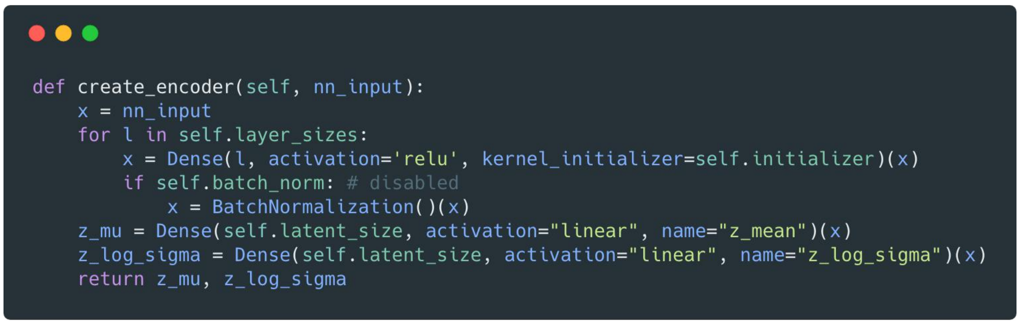

```python
def vae_loss(self, x, x_decoded_mean):
    regularizer = 1.0
    return -K.mean(self.recon_loss(x, x_decoded_mean) + regularizer*self.kl_loss(x, x_decoded_mean))

def recon_loss(self, x, x_decoded_mean):
    return -0.5*losses.mean_squared_error(x, x_decoded_mean)

def kl_loss(self, x, x_decoded_mean):
    return 0.5 * K.sum(1 + self.log_sigma - K.square(self.mu) - K.exp(self.log_sigma), axis=1)
```

Figure D.2: The VAE has two losses: the reconstruction loss and the KL divergence between the standard normal distribution and the normal distribution sampled with the output parameters from the encoder.

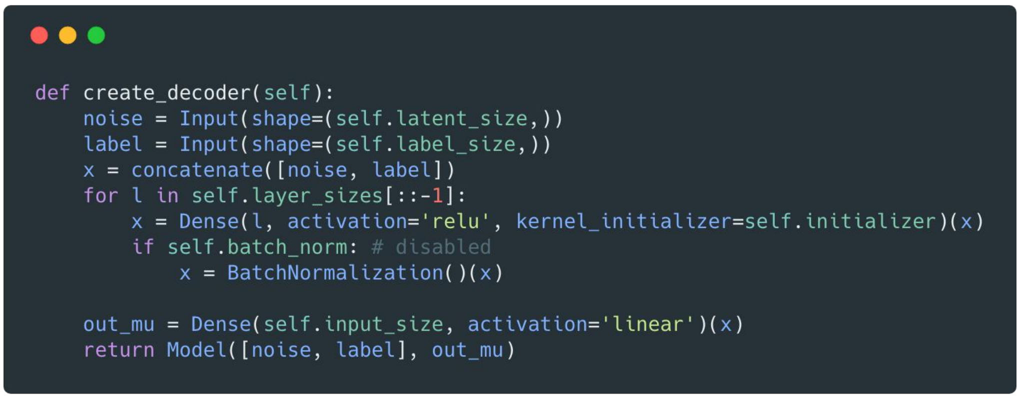
The encoder is a simple 2-layer `ReLU` network with (256, 128) units per layer. The network accepts the features (costates and cost) and the labels (initial and final states) as inputs, which are concatenated to a single tensor. This is the suggested method for conditioning the model on the labels [14]. The encoder outputs the the parameters for a normal distribution μ and σ^2. The decoder network has the same structure with the amount of units flipped. The output of the decoder is set to linear as constraining them to the domain of the variables is detrimental. Furthermore, the decoder samples a standard normal distribution from which it attempts to recreate the input. The sampling is provided by a `Lambda` layer to ensure that back-propagation is possible through a random distribution. Code snippets for the encoder and decoder are presented in Figures D.3 and D.4, respectively.

Training for the VAE is handled directly by the `fit()` function from Keras, which minimises the total loss. The choice for optimiser is `RMSProp` with the default learning rate of `1e-3`, as suggested by the reference design. Other optimisers have been tested and do not improve performance. The batch size is set to 100. Training is terminated after 30 epochs once the loss has converged.



```python
def create_encoder(self, nn_input):
    x = nn_input
    for l in self.layer_sizes:
        x = Dense(l, activation='relu', kernel_initializer=self.initializer)(x)
        if self.batch_norm: # disabled
            x = BatchNormalization()(x)
    z_mu = Dense(self.latent_size, activation="linear", name="z_mean")(x)
    z_log_sigma = Dense(self.latent_size, activation="linear", name="z_log_sigma")(x)
    return z_mu, z_log_sigma
```

Figure D.3: The encoder network encodes the samples from the dataset to the parameters of a normal distribution.



```python
def create_decoder(self):
    noise = Input(shape=(self.latent_size,))
    label = Input(shape=(self.label_size,))
    x = concatenate([noise, label])
    for l in self.layer_sizes[::-1]:
        x = Dense(l, activation='relu', kernel_initializer=self.initializer)(x)
        if self.batch_norm: # disabled
            x = BatchNormalization()(x)

    out_mu = Dense(self.input_size, activation='linear')(x)
    return Model([noise, label], out_mu)
```

Figure D.4: The decoder network attempts to reconstruct the input from a standard normal distribution.

D.2.2. GAN Implementation

The implementation of GAN is a remedy to the poor learning performance of VAE in the context of Generative CoLearn. Similar to VAE, an open-source online implementation[2] is used as a reference design. Note that the GAN implementation is done directly in Tensorflow as the implementation in Keras only results in convoluted code and suboptimal performance.

The main two components of GAN are the generator $G(z)$ and the discriminator $D(x)$, the adversarial networks. Both networks play a *minimax* game where they try to beat each other in their objective. The generator aims to generate samples from a random distribution that looks as real as possible to fool the discriminator. The discriminator on the other hand, tries to discern real samples from generated samples. The feedback signal from the discriminator is used to improve the generated samples, resulting in an *implicitly* learned generator distribution, represented by $G(z)$. The GAN objective is formalised by

$$\min_G \max_D V(D,G) = \mathbb{E}_{x \sim p_{data}(x)}[\log D(x)] + \mathbb{E}_{z \sim p_z(z)}[\log(1 - D(G(z)))]. \tag{D.3}$$

As a result of this adversarial objective, the original formulation of GAN is notoriously difficult to optimise. Various attempts have been made in literature to stabilise GAN training. A recent large scale study presented several popular adversarial objectives (loss functions), which improve sampling quality and stability during training [15]. From the proposed losses, the least-squares loss function is implemented in this research for three reasons. First, the implementation is very straightforward and does not require loss regularisation as with other proposed objectives, resulting in fast training times and reduced complexity. Second, the quality of the generated samples is very much on par with the other objectives. Finally, the authors of the least-squares GAN propose a method to condition the networks, which is not the case for most methods found in the study. Therefore, the implementation for the least-

[2]`https://github.com/wiseodd/generative-models/`

squares GAN is warranted. The formulation of the conditional least-squares GAN is given by Equation (D.4).

$$\begin{aligned} \min_D V(D) &= \frac{1}{2}\mathbb{E}_{x\sim p_{data}(x)}[(D(x|y)-1)^2] + \frac{1}{2}\mathbb{E}_{z\sim p_z(z)}[D(G(z|y))^2] \\ \min_G V(G) &= \frac{1}{2}\mathbb{E}_{z\sim p_z(z)}[(D(G(z|y))-1)^2] \end{aligned} \tag{D.4}$$

Implementation of both networks is straight forward. The generator, uses a fairly deep network with 5 `ReLU` layers and each (32,64,128,256,512) units, which gives the best results for the pendulum dataset. The latent space z has 32 dimensions. The inputs are the labels and noise sampled as $z \sim \mathcal{U}(-1, 1)$, concatenated as a single tensor. This concatenation step conditions the network on the labels allowing it to generate conditional result [16]. The output of the generator is constrained to the domain of the variables by means of activation functions. For the costates `tanh` is used, whereas for the cost `ReLU` is used to constrain the output. The discriminator uses 5 hidden Leaky `ReLU` layers with the order of units flipped. The inputs for the discriminator are the true or generated samples and the corresponding labels. For conditioning, both the sample and label tensors are concatenated. After a batch normalisation step each layer is conditioned additionally on the labels as suggested by [17]. The output layer has a single unit with linear activation. Generator samples that look fake will cause the discriminator to output a negative value while real looking samples will output a positive value. The decision boundary is at 0. The networks are visualised in Figure D.5.

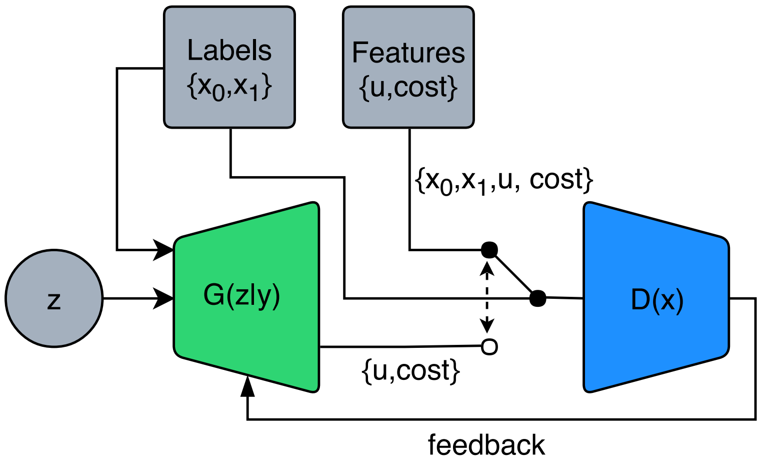


Figure D.5: The objective of the two adversarial networks is to outcompete each other. The generator tries to fool the discriminator, which in turn tries to discern true from generated samples.

The training for GAN is performed manually, in contrast to the VAE. Both networks are optimised using the `Adam` optimiser with $\beta_1 = 0.5$ while keeping the other hyper-parameters set to their default value. The batch size is 100. First, the discriminator is trained by showing it real samples and generated samples from the generator. The weights of *only* the discriminator are updated. Next the generator is trained by generating samples and showing only those to the discriminator. The output signal of the discriminator is used to update the weight of *only* the generator. These operations account for a single epoch. In total, the GAN is trained for 30000 epochs. The goal for the GAN is that both networks reach an equilibrium value. However, this does not mean training has converged. In fact, at this point both networks are performing equally and are improving each other. This is one of the reasons why training GAN is so difficult as it is generally unknown when training has converged. Typically, deep learning practitioners rely on generating images for inspection during training and terminating it if the samples meet some subjective requirement. However, this inspection is not trivial for real valued samples as is the case in Generative CoLearn. The next section will detail how the learning performance of the models is evaluated.

D.3. Benchmarking

During development of a DGM it is important to quickly receive feedback on its performance. Waiting until the training has converged to generate samples or testing the performance online during RRT, is impractical due to long training times, which is especially the case for GAN. Furthermore, for GAN it is not known when training has converged as both networks improve while their loss is in equilibrium. Therefore, two benchmarks were devised to give insight in the training dynamics of the DGM and the quality of the generated samples. The first benchmark uses a popular dataset used throughout machine learning. The second benchmark uses the pendulum dataset and visualises how well the DGM can uncover features known a priori. During training, figures are generated to give insight in the learning performance. The content of the figures depends on the dataset. The DGMs that are benchmarked next are the conditional variants of the VAE and GAN.

D.3.1. MNIST Validation

Initially, the correctness of implementation is tested with MNIST[3], which is generally the default benchmarking dataset in machine learning. The dataset contains labeled 28×28 images of handwritten digits from 0-9. In unsupervised learning the DGM simply learns to generate random real-looking digits. However, it is also possible to condition the DGM on labels. This is done by converting the labels to one-hot encoding, which is essentially a boolean vector for the digit value. For example, the number 2 would become `[0,0,1,0,0,0,0,0,0,0]`. This vector is concatenated to the target data. In the case of MNIST this is a vector of size 784 resulting from reshaping the original matrix to a single dimension. Using this benchmark gives us the ability to see whether the models are being conditioned properly as this is difficult to do with the real valued datasets. During training the models generate digits from 0-9. As soon as the digits are in the right order and look realistic, as shown in Figure D.6, the training is terminated, confirming the success of implementation.

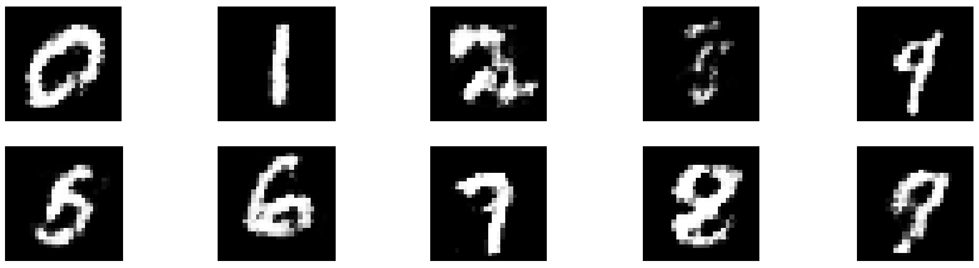


Figure D.6: Generated digits by the GAN verifies that implementation of the conditional modification is successful. These are results after only 30000 epochs and learning errors are still clearly visible in digits 2,3 and 4.

D.3.2. Pendulum Dataset Validation

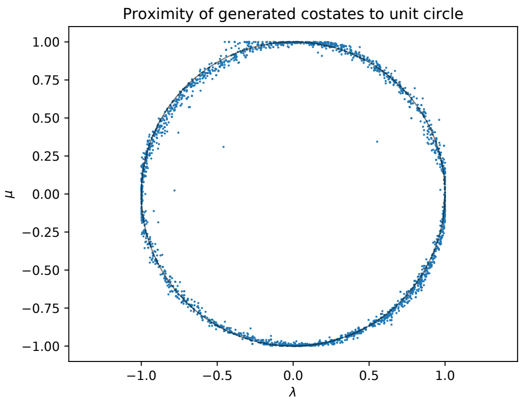
Following the validation of the model implementation, the DGMs are trained on the pendulum dataset to determine which performs better. Unlike MNIST, it is not trivial to generate meaningful samples for determining the preferred DGM, as the data consists of real valued numbers. This makes it increasingly difficult to properly tune the hyper-parameters since feedback during training is limited. However, several features are known a priori, such as the data distribution and that the costates are to be sampled from the unit sphere. Instead of relying solely on the raw data output by the DGM, we use the data to generate figures in Python and observe whether the output matches our expectation on these known features. This can give us insight in the learning performance of the DGM both during and after training.

The GAN and VAE generate costates and cost by querying them with labels from the test set, which

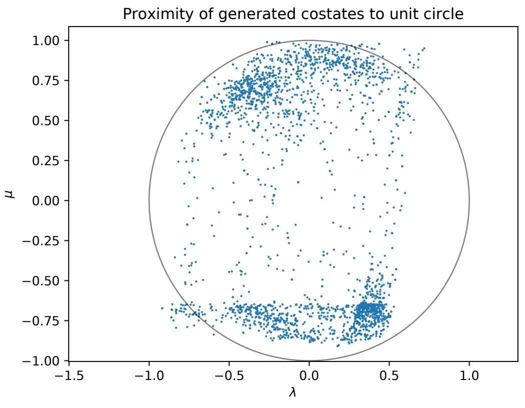
[3]`http://yann.lecun.com/exdb/mnist/`

consists of 20% of the total dataset (8000/40000). Recall that for a simple pendulum, the learning data consists out of 4 labels ($\theta_{\text{init}}, \omega_{\text{init}}, \theta_{\text{final}}, \omega_{\text{final}}$), containing the initial and final states and 4 features ($\lambda, \mu, \text{cost}, t_f$), containing the costates, cost and simulation time. The generated samples are used to create the aforementioned figures. The figures shown in this section have been generated *after* the optimal hyper-parameters have been found. However, the methodology of these figures was used to tune the DGMs.

Initially, the generated costates are used to observe if and how well they are generated from the unit circle, shown in Figure D.7. It is clear that the VAE is struggling to uncover that the costates should lie on the unit circle.



(a) GAN



(b) VAE

Figure D.7: The GAN is able to generate costates that lie on the unit circle while CVAE struggles to uncover this feature.

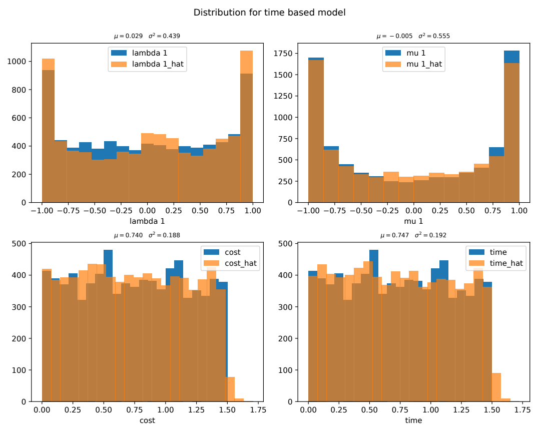
Additionally, a similar discrepancy in performance appears when the generated and true data distributions are visualised by means of histograms. For each feature, the complete test set is used as the ground truth and as queries for generating samples. In Figure D.8 it is clear that GAN outperforms VAE. The GAN is able to match the modes in the true data distribution, seen as the edge peaks in the distribution of the costates. The VAE struggles to match the distribution.

As a result, it can be concluded that the GAN outperforms VAE, specifically in the domain of Generative CoLearn. The GAN is able to generate costates that lie on or very close to the unit circle. Furthermore, GAN is superior in capturing the modes of the data distribution. The VAE is not able to sufficiently perform in this benchmark and as such is not subject to further evaluation. The next step is to compare GAN with KNN, which is the learning algorithm used in RRT-CoLearn and is discussed next.

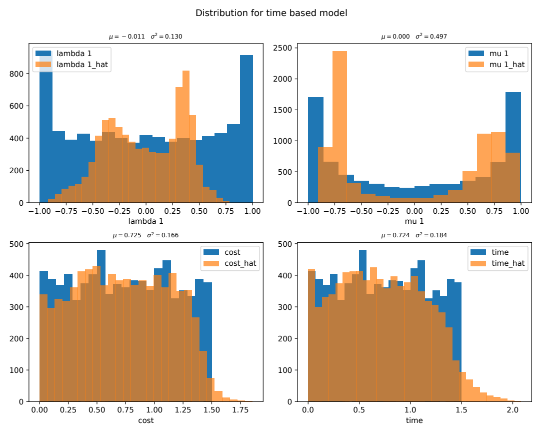
D.4. Model Analysis

The learning algorithm in Generative CoLearn is GAN and is used for predicting steering inputs and cost. In contrast, RRT-CoLearn uses KNN as the learning algorithm for prediction. Since Generative CoLearn is an extension of RRT-CoLearn, it is necessary to compare both learning algorithms for benchmarking purposes. Therefore, a variety of tests are performed with the pendulum dataset. Additionally, the planar arm dataset is used for further analysis.

Initially, the KNN algorithm is subject to the same two tests of the benchmark described in the previous section. The first test involves the proximity of costates to the unit circle by generating costates from 1000 query trajectories from the test set. From Figure D.9a it is clear that KNN is not able to generate costates from the unit circle. Since KNN is a non-parametric learning algorithm, it simply stores the data in a tree. Once the algorithm is queried with a trajectory, it looks up k neighbours that are the closest to the query in terms of Euclidean distance. These neighbours are averaged and returned as a prediction. Since the nearest neighbours all lie on the unit circle, the predicted costates tend to end up within the unit circle. A clear representation can be seen in Figure D.9b. The samples from the

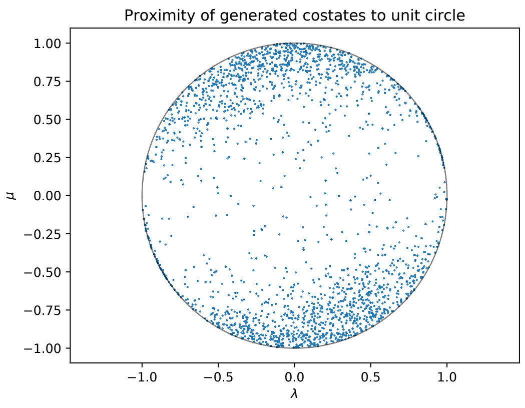


(a) GAN

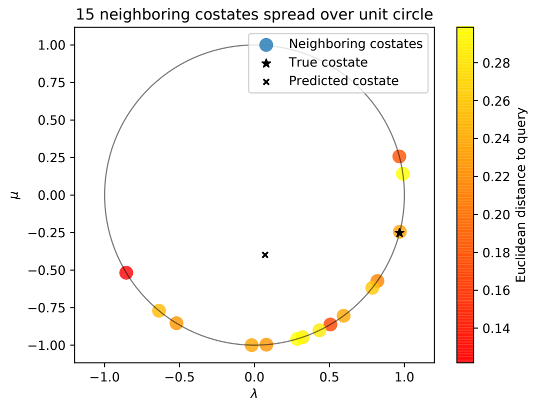


(b) VAE

Figure D.8: The GAN is able to capture the multimodal nature of the data, visible as the edge peaks in the costates (D.8a). The VAE attempts to capture the edge peaks, but is not able to properly match the true data distribution (D.8b).



(a) 1000 generated costates by KNN. Interpolation results in costates inside the unit circle.



(b) Costate interpolation with 3 nearest neighbours. 15 nearest neighbours are shown.

Figure D.9: KNN is struggling to generated the costates from the unit circle.

test set that lie nearest to the ground truth on the unit circle are not necessarily the nearest in terms of Euclidean distance. The three darkest circles represent the costates of the nearest sample to the query. Interpolating over these nearest costates results in the predicted costate to lie within the unit circle. This prediction is not necessarily invalid due to the parametrisation of the costates and the 1-dimensional nature of the problem. However, these type of predictions may not be appropriate for higher dimensional problems.

The second test involves the visualisation of the true and generated data distributions. From Figure D.10 it is clear that the generated distribution for the costate λ does not match the true distribution. This is also easily observed in D.9a, where the majority of the points lie near $\lambda = 0$ for $\mu \pm 1$.

Additionally, we are interested in finding how well both GAN and KNN can generalise. From the test set, 3 random samples are selected from which the trajectories are used to query the learning algorithm. For each query, 1000 costates are generated and plotted. As expected from KNN, shown in Figure D.11b, the generated costates lie on the same location and generally within the unit circle, attributed to the deterministic nature of the algorithm. However, GAN is able to generate multiple costates for a single trajectory. This is generally the case when the trajectory does not involve a switch in the sign of μ. If a query trajectory involves a switch in μ, then λ is sampled accordingly. The fact that GAN is able to uncover this relationship between trajectory and torque is quite remarkable. This effect can be observed in Figure D.11a where the costates that involve a switching torque are all concentrated near

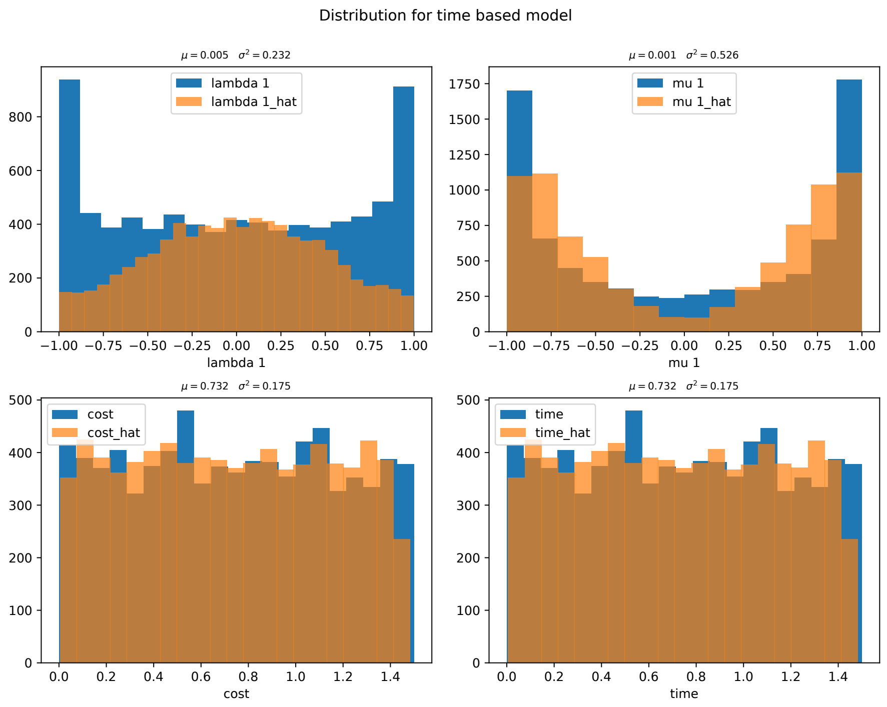
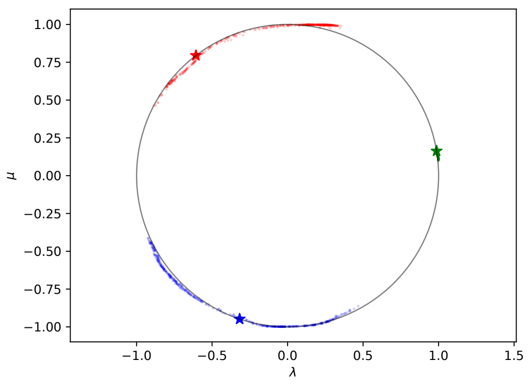


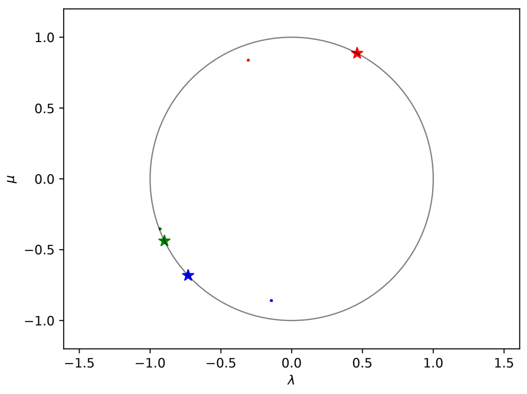
Figure D.10: KNN is able to nearly match the distribution for μ, yet falls short in the distribution for λ.

the green star.

A more in depth approach in analysing the performance of the learning algorithms is to use the generated costates in a forward simulation. From the test set 1000 samples are selected. Recall that these samples contain $(\theta_{\text{init}}, \omega_{\text{init}}, \theta_{\text{final}}, \omega_{\text{final}})$ and $(\lambda, \mu, \text{cost}, t_f)$. The trajectories are used to query both KNN and GAN to generate costates and cost. The difference between true and predicted cost is measured as the mean squared error (MSE) between the two values, shown in Table D.1. Furthermore, the initial state, generated costates and cost are fed into a forward simulation. Since we already know the final state from the test set, we can compare it to the final state provided by the forward simulation using the generated costates. The MSE is computed between the *target* and *actual* final state and is summarised in Table D.1. We see that the GAN slightly outperforms KNN. The relatively high error is still not fully understood, but it is hypothesised that it is attributed to the low state-space coverage of only 40000



(a) GAN



(b) KNN

Figure D.11: GAN is able to generate multiple costates for a single query. Due to the deterministic nature of KNN, all the generated costates lie on the same location for each query.

Table D.1: A summary of the model analysis for GAN and KNN.

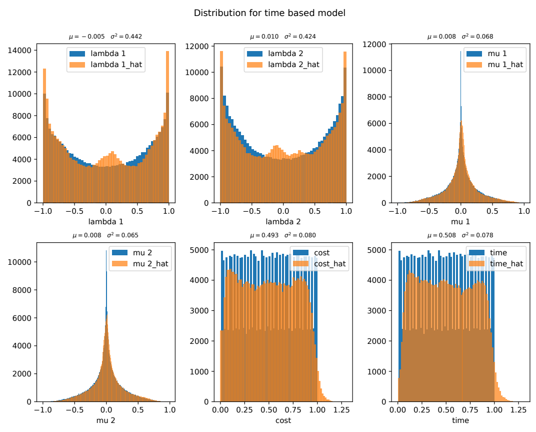
	DoF	Proximity	θ_E	ω_E	Cost$_E$	t_{single}	t_{batch}
GAN	1	$1.0 \pm .09$	.240	.713	.006	0.4 ms	63 ms
KNN	1	$.85 \pm .50$	.250	.723	.012	0.3 ms	65 ms
GAN	2	$1.0 \pm .10$	.006	.049	.005	0.4 ms	1.9 s
KNN	2	$0.7 \pm .21$	.008	.031	.016	0.4 ms	18.0 s

samples. Next, the performance of GAN is investigated for the planar arm.

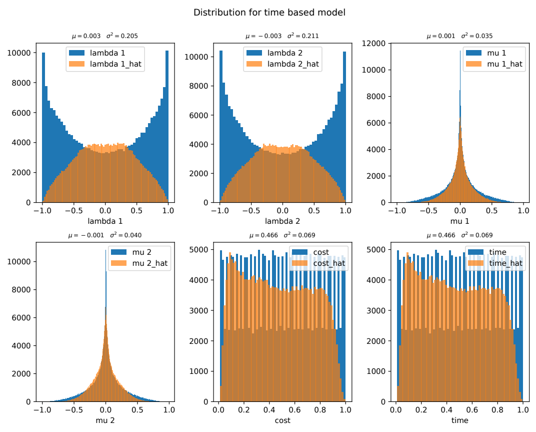
Visualisation of the costates for a pendulum is simple, yet is impractical for a system with more degrees of freedom. If we take the planar arm, for example, the costates are sampled, and thus generated, from a 4D-sphere, which is difficult to visualise. As a result, the norm of the costates is plotted in a histogram. This is justified by the fact that the true costates are always sampled as points from a unit n-sphere. This requires that the norm of the costates to be equal to unity. Therefore, the generated costates should *approach* unity. A trained GAN on the planar arm dataset generates costates near the unit sphere, of which the histogram is shown in Figure D.13a. The KNN algorithm does not generate the costates from the unit sphere as the distribution of their norm varies wildly, shown in Figure D.13b. Additionally, the feature distribution is shown in Figure D.12 for both GAN and KNN. Similar to the pendulum dataset, the GAN is able to properly match the true data distribution, while KNN is struggling with some distributions. Forward simulations with generated costates are also performed of which the results are tabulated in Table D.1. The probable reason why the errors for the planar arm are lower than the pendulum is due to the size of the dataset. Since the dataset contains 1.1 million samples, there is a much higher state-space coverage and both algorithms are able to effectively learn from many more examples, resulting in more accurate predictions.

Finally, the prediction time for KNN and GAN is measured. For real-time RRT applications, prediction time is critical, especially when the amount of nodes, and as such the required predictions increase. Prediction time is measured for both the pendulum and planar arm dataset by querying a single sample or a batch of samples, resulting in t_{single} and t_{batch}. The reported measurements are median values over 10 epochs. The batch of samples contains 8000 and 240000 samples for the pendulum and planar arm, respectively. For the pendulum we find that both KNN and GAN perform almost identically. However, for the planar arm a much larger batch size is used and the difference between the two algorithms becomes apparent. Even though the prediction time for a single sample is 0.4 ms for both GAN and KNN, the large batch size is detrimental for KNN. The GAN is almost an order of magnitude faster. The results are summarised in Table D.1.

In this section the machine learning aspect of Generative CoLearn was further substantiated regarding the original material presented in the conference paper. The goal was to find a deep generative model that could compete against KNN for predicting cost and steering inputs for RRT. Out of VAE and GAN, the two most popular generative models, the GAN proved to be much more capable in the domain of Generative Colearn. Further analysis revealed that the GAN outperforms KNN as well, and has excellent generalisation capabilities for both the pendulum and planar arm datasets. In the next section, we will find how these results affect RRT performance and why it is not practical to use KNN for kinodynamic RRT with higher DoF systems.

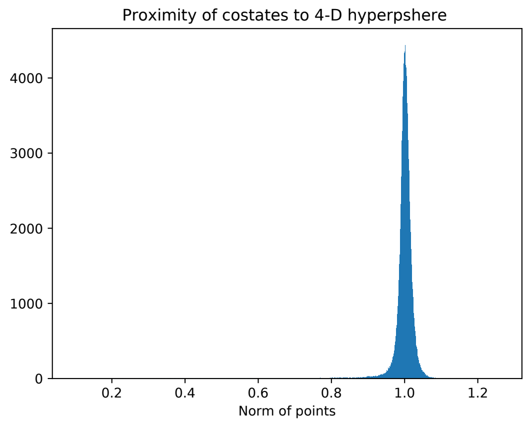


(a) GAN

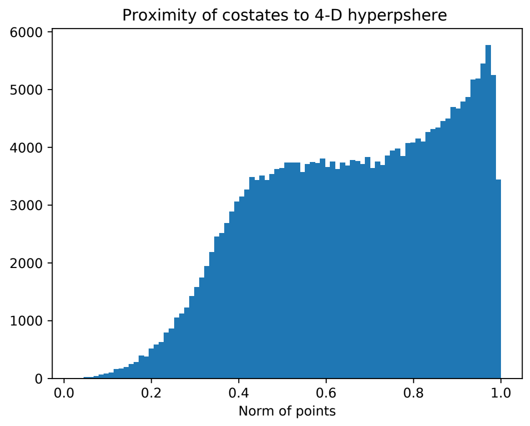


(b) KNN

Figure D.12: For the planar arm, the GAN is able to properly resolve and match the data distribution of the training data with only minor errors. The KNN algorithm is struggling to match the distribution for λ and the cost.



(a) GAN



(b) KNN

Figure D.13: The true costates are always sampled from the unit n-sphere. Therefore, most of the generated costates should lie near unity.

E
Planner

The computational difficulty of kinodynamic planning is generally alleviated with sampling based methods. The most popular of such methods is Rapidly-exploring Random Trees (RRT) and allows for effective exploration of the planning space, i.e. state space, in kinodynamic planning. The implementation of the RRT algorithm is straight forward and general and as such only notable deviations from the algorithm will be discussed in this section. The main RRT algorithm used in Generative CoLearn is outlined in Algorithm 1.

E.1. Reachability

A key component in Learning-RRT algorithms, is selecting a set of nodes from which the randomly sampled state can be reached, aptly called *reachability*, which is visualised in Figure E.1. Reachability is based on the notion that not all randomly sampled nodes are dynamically feasible [3]. For example, robots have both dynamic constrains (i.e. maximum torque) and kinematic constrains (i.e. maximum angle). Therefore, some target states are not reachable[1] from an initial state, exemplifying the necessity of a reachability check. As this is a difficult to compute problem, real-time performance in kinodynamic applications is hindered, necessitating approximate methods. In the case of Generative CoLearn, the limitation of reachability exposes itself in the generalisation capability of the learning algorithm and is caused by two inherent limitations. First, standard machine learning algorithms, which include GAN and KNN, have poor extrapolation abilities resulting in poor predictions for queries that lie outside of the region the training data was sampled from [18]. Second, even if the query lies within the sampled region, sufficient examples are necessary for the learning algorithm to make a meaningful prediction. In the case of KNN, the prediction is a result of linear interpolation in Euclidean space, which is not necessarily valid. For the GAN, the interpolation occurs on some learned data manifold, which has embedded the non-linear relationship between samples. However, if the amount of samples is small, important modes can be missed, still resulting in invalid predictions.

As a result of this limitation in machine learning, only queries that will yield meaningful predictions should be considered. In Generative CoLearn two types of reachability methods are implemented. The first type is the Euclidean reachability, which is a simple method as is used in RRT-CoLearn. Second is discriminative reachability, which is a novel approach where the trained discriminator of the GAN is used for classification of feasible trajectories. First we discuss Euclidean reachability.

[1]Note the difference between feasibility and reachability: a trajectory is *feasible* if the target node is *reachable* from the corresponding node in the tree.

Algorithm 1 Generative Colearn

```
data ← generate_data()
$G, D$ ← train_network(data)
node_tree ← $x_{\text{start}}$
while not goal() do
    $x_{\text{random}}$ ← random_state()
    $N_{\text{reachable}}$ ← empty()
    for $i$ in node_tree do
        $T \leftarrow \{x_i, x_{\text{random}}\}$
        $\hat{u}$, cost ← $G(T)$
        if $D(\hat{u}, T) > 0$ then
            append($N_{\text{reachable}}$, $\{x_{\text{random}}, \hat{u}, \text{cost}\}$)
        end if
    end for
    $x_{\text{near}}, \hat{u}$ ← min_cost($N_{\text{reachable}}$)
    $x_{\text{final}}$ ← simulate($x_{\text{near}}, \hat{u}$)
    append(node_tree,$x_{\text{final}}$)
end while
```

E.1.1. Euclidean Reachability

Once a random node has been sampled, a set of trajectories is generated from each node in the current tree. This is equivalent to creating a matrix T, containing entries $(\theta_i, \omega_i, \theta_{\text{rand}}, \omega_{\text{rand}})$ for i nodes in the tree. The feasibility of each trajectory in T needs to be determined by checking its similarity to what has been seen during training of the learning algorithm. Similar to RRT-CoLearn, this is implemented by fitting a *separate* KNN model with only the trajectories from the training set and performing a nearest neighbour lookup. Trajectories that are within a Euclidian distance of δ_{max} to two nearest neighbours are considered feasible. The nodes in the tree corresponding to the feasible trajectories are stored as a reachable set $N_{\text{reachable}}$. For each node in the set, the steering and cost is predicted by the learning algorithm. The node with the lowest cost is expanded by running a forward simulation. This process is visualised in Figure E.1.

The problem with Euclidean reachability ultimately lies in scalability. Aside from tuning δ_{max} being cumbersome, performing a nearest neighbour lookup is prohibitively expensive as the data dimensionality grows. Therefore, using the discriminator as a classifier for feasible trajectories is an appropriate replacement for the reachable bound.

E.1.2. Discriminative Reachability

Another approach to reachability is to use the trained discriminator as a classifier for feasible trajectories. This approach is eliminates the tuning of the reachable bound δ_{max} and is much faster than a nearest neighbour look-up. The latter is impractical for real-time applications as the dimensionality of the dataset grows. This has been observed from preliminary benchmarks in Appendix D. This section will discuss the implementation of this approach and is outlined in Algorithm 1.

The discriminator is trained to discern generated samples from real samples originating from the dataset. Additionally, the generator is trained to generate samples that are real with feedback from the discriminator. Therefore, the following assumptions are made.

1. The generator will generate **realistic** samples for queries that are similar to those seen during during training, resulting in feasible trajectories.
2. The generator will generate **unrealistic** samples for queries that are not similar to those that have been seen during training, resulting in trajectories that are **not** feasible.
3. The discriminator will classify the generated samples to the correct side of the decision boundary based on assumptions 1 and 2.

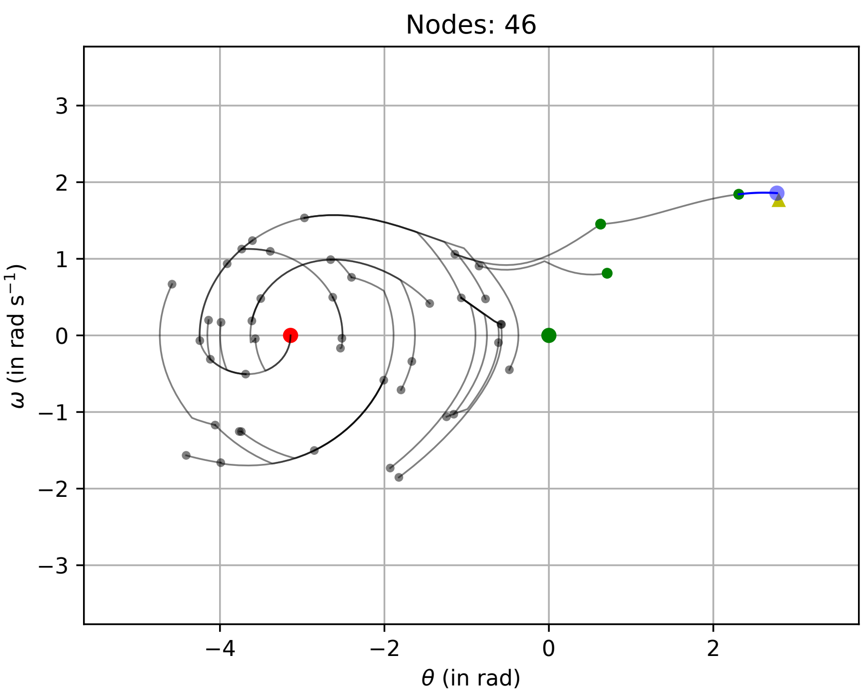


Figure E.1: An example of reachability for the pendulum is presented. A random state (yellow triangle) is only reachable from a set of nodes (green dots) that result in feasible trajectories. From that set the node with the lowest cost is selected for expansion. The trajectory resulting from forward simulation with predicted costates is shown in blue.

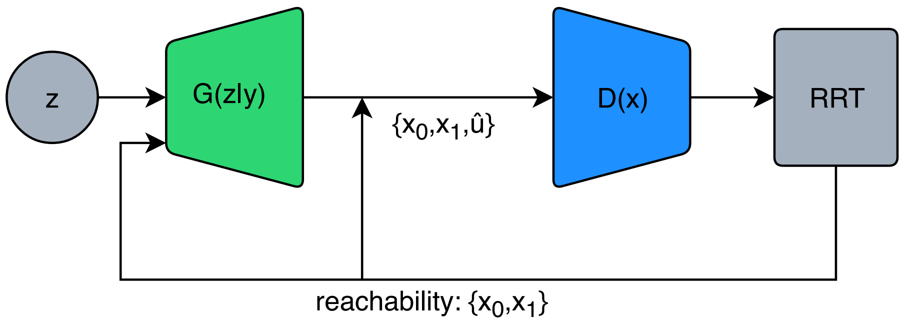


Figure E.2: The discriminator classifies the reachability of a set of trajectory queries from RRT based on the generated costates.

Similar to the Euclidean reachability, a set of trajectories is generated from the nodes in the tree to the randomly sampled node. These trajectories are fed into the generator and then directly into the discriminator for classification, as shown in Figure E.2. The output of the discriminator is a linear value with 0 being the decision boundary. If the generated costates with the associated trajectory result in a positive output, then the trajectory is regarded as feasible and the corresponding node in the tree is added to $N_{\text{reachable}}$. Otherwise, the discriminator outputs a negative value and the trajectory is disregarded. If an empty set of reachable nodes is returned, a new random node is sampled. The discriminative reachability approach is much faster than Euclidean reachability and does not require tuning δ_{max}.

E.1.3. Determining the Reachable Bound

Only trajectories that fall within the reachable bound δ_{max} are considered feasible. However, determining the value for δ_{max} is not trivial as there is no heuristic. The most straightforward method is to run the RRT algorithm 100 times for values of δ_{max} sampled from `linspace(0.1,1,10)`. Due to the computational cost, this method can only be performed on the pendulum dataset. Three metrics are collected and visualised in Figure E.3. The failure rate indicates when the RRT run is terminated after reaching 1000 nodes without convergence. The node count and steering error are straightforward and are the main performance metrics. It is clear that the reachable bound has an effect on both learning algorithms. As the value for δ_{max} increases, the amount of nodes and steering error for both learning algorithms increases as well. However, the key is that GAN remains relatively stable across all three metrics, exemplifying its robustness. With this visual aid, it is now easy to determine the value for δ_{max} for both learning algorithms. We find for GAN, $\delta_{\text{max}} = 0.4$ and for KNN, $\delta_{\text{max}} = 0.3$. These values are chosen based on the most balanced performance between all three metrics.

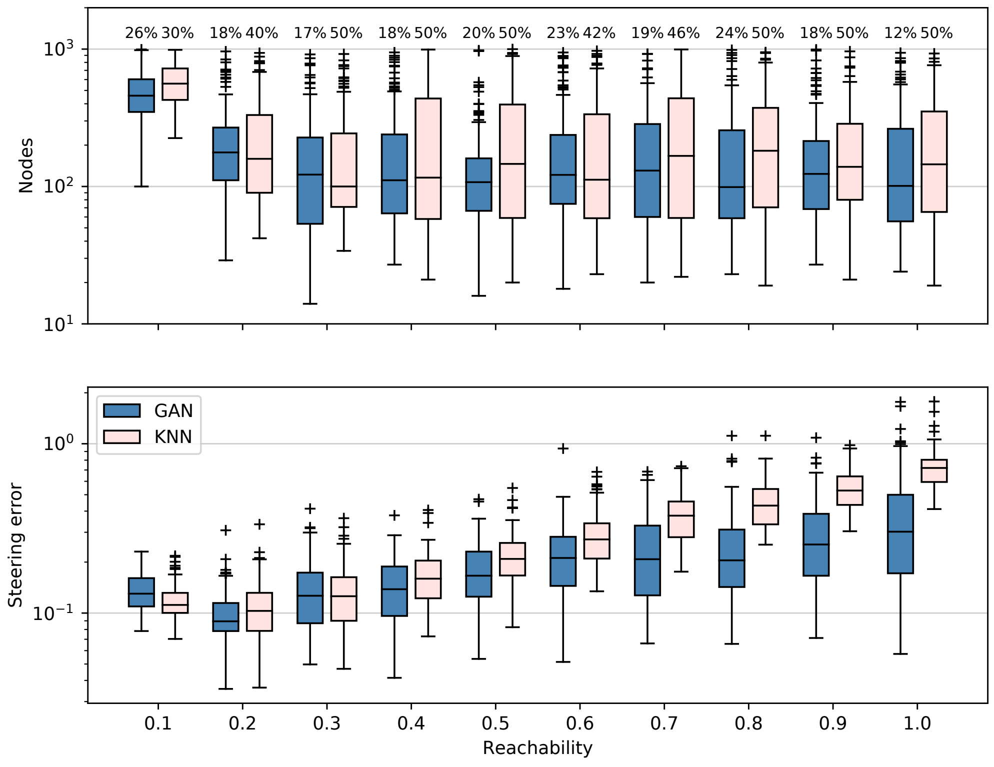


Figure E.3: Reachability has an impact on the performance of both learning algorithms. However, KNN is affected more than GAN. The failure rate is indicated as a percentage on top of the boxes.

E.2. Pendulum Results

Most of the analysis of the pendulum results have been presented in the conference paper. A summary of the RRT performance is given in Table E.1. Even though KNN has a slightly better error and planning time, GAN is much more robust and requires far fewer nodes for successfully planning a path. The discrepancy in planning time is attributed to the prediction time, which is slightly higher for GAN. However, GAN is invariant to data dimensionality in terms of prediction time and is far less sensitive to large batch sizes as opposed to KNN.

Additionally, this section provides insight in the expansion of the nodes during swing-up. In Figure E.4 various examples are shown in the initial stages of RRT on how the tree is expanded by GAN. Note that these results do not differ much from KNN and simply serve as a visual aid. The GAN is able to predict steering inputs accurately resulting in small errors. More remarkably, trajectories are predicted that contain a switch in torque and still result in the final state to be near the target state. It is interesting to note that the selected node for expansion is not always the nearest node in Euclidean space. This is an indication that cost prediction works appropriately.

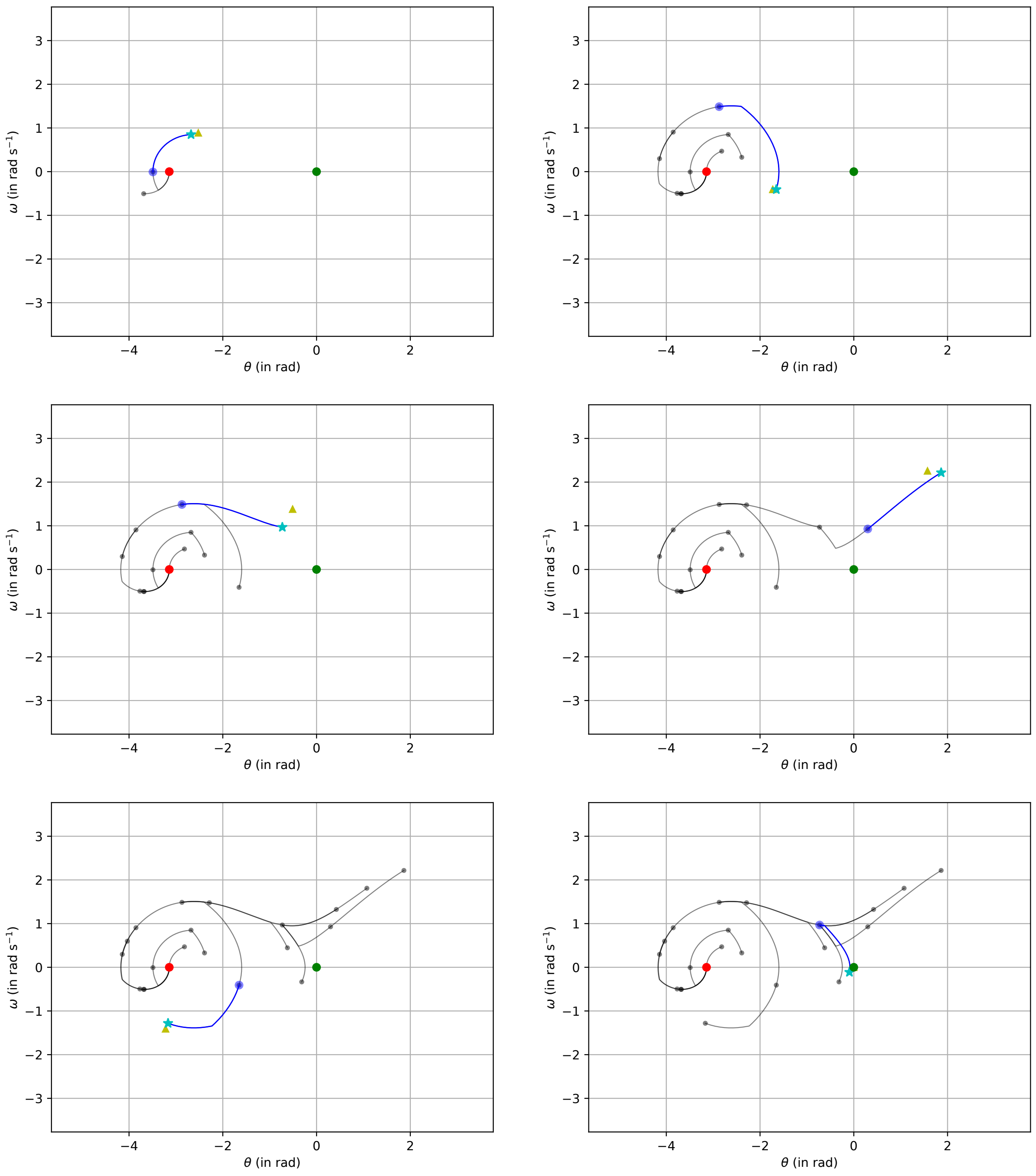


Figure E.4: The GAN is able to accurately predict costates to expand the nearest node. Note that some expansions include a switch in torque which can be seen as trajectories with sharp turns between nodes. The yellow triangle indicates the target state. The cyan star indicates the true final state and sometimes overlaps the yellow triangle indicating a near perfect prediction. The path taken is indicated with the blue line.

Table E.1: For the pendulum dataset, GAN has better performance in terms of fail rate and node count. KNN is slightly faster and accurate.

	Error	Time	Nodes	Fail
GAN	0.15	0.85 s	92	13%
KNN	0.11	0.81 s	132	50%

E.3. Planar Arm Results

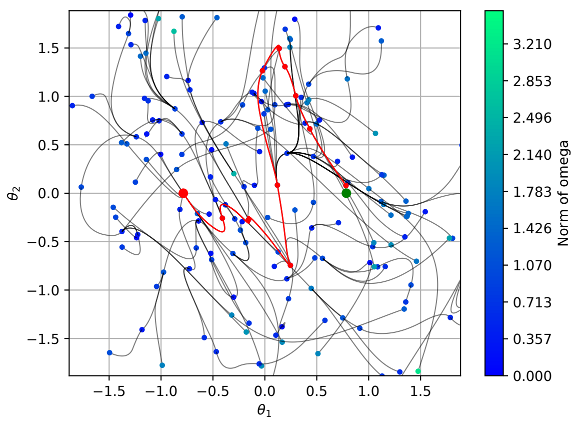
The goal of Generative CoLearn is scaling to systems with higher degrees of freedom. There still remains an uncertainty on how much data is necessary for a learning algorithm to generalise well. As such, a simple planar arm is tasked to move from $(\theta_0, \theta_1, \omega_0, \omega_1) = (-\frac{\pi}{4}, 0, 0, 0)$ to $(\frac{\pi}{4}, 0, 0, 0)$. This relatively simple task is chosen as a proof of concept to demonstrate that Generative CoLearn works for systems with more degrees of freedom. More complex tasks, e.g, that include gravity, are also possible by modifying the equations of motion from Appendix C and is left for future work.

From the previous sections, we have learned that KNN does not scale well with data dimensionality. As the node tree grows during RRT, the KNN algorithm becomes slower in prediction, and more critically, in finding a reachable set of nodes. This speed deterioration makes KNN impractical for larger scale planar arm experiments as previously performed for the pendulum. Initial tests have indicated that with a threshold of 1000 nodes, the algorithm never converged in 10 attempts. Each attempt lasted about 10 minutes, becoming slower as thee tree grew. This warrants the disregard of KNN for experiments on the planar arm. Instead, only the GAN is considered for the planar arm experiments, of which the results are summarised in Table E.2. With the large amount of data and incorporated augmentation, GAN is very robust with near constant convergence and low node count. The high planning time is caused by the suboptimal implementation.

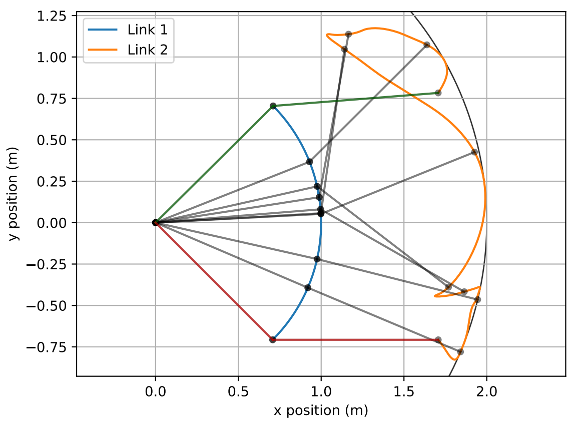
Table E.2: Only GAN is considered for the planar arm experiments as KNN combined with reachability is simply too slow.

	Error	Time	Nodes	Fail
GAN	0.16	8.80 s	94	0.1%

Visualisation of high dimensional parameter- and state-space is impractical. The path taken by the planar arm is therefore shown in an augmented state-space representation. In Figure E.5a, the axes represent the angles of the joints (θ_1, θ_2) and the colour of the nodes represents the norm of the angular velocities (ω_1, ω_2). This augmented view gives insight in the path taken by the arm in terms of angles. The most logical path would be a straight line across $\theta_2 = 0$, where the second link does not move. However, the classical RRT algorithm does not guarantee an optimal trajectory [19]. This is especially obvious when the path is visualised in configuration space, shown in Figure E.5b. The planar arm initially moves away from the target and follows a peculiar and very non-optimal path, after which it overshoots the target. However, the algorithm ultimately converges to the goal, as can be seen in the figure. In order to achieve optimal trajectories, planning algorithms that guarantee that optimality are required, such as RRT*.



(a) Augmented state space



(b) Euclidean space

Figure E.5: RRT does not guarantee an optimal path. In (E.5a) the most logical trajectory would be along $\theta_2 = 0$. A task-space representation of the path in (E.5b) shows how much the path can deviate from optimality. The red line indicates the start state, the green line indicates the goal state.

Bibliography

[1] W. J. Wolfslag, M. Bharatheesha, T. M. Moerland, and M. Wisse, “RRT-CoLearn: Towards Kinodynamic Planning Without Numerical Trajectory Optimization,” *IEEE Robot. Autom. Lett.*, vol. 3, pp. 1655–1662, jul 2018.

[2] J. Canny, J. H. Reif, B. R. Donald, and P. G. Xavier, “On the complexity of kinodynamic planning,” 1988.

[3] R. E. Allen, A. A. Clark, J. A. Starek, and M. Pavone, “A machine learning approach for real-time reachability analysis,” in *2014 IEEE/RSJ Int. Conf. Intell. Robot. Syst.*, pp. 2202–2208, IEEE, sep 2014.

[4] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, “Generative Adversarial Nets,” in *Adv. Neural Inf. Process. Syst. 27* (Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, eds.), pp. 2672–2680, Curran Associates, Inc., 2014.

[5] O. Mogren, “C-RNN-GAN: A continuous recurrent neural network with adversarial training,” in *Constr. Mach. Learn. Work. NIPS 2016*, p. 1, 2016.

[6] Z. Erickson, S. Chernova, and C. C. Kemp, “Semi-Supervised Haptic Material Recognition for Robots using Generative Adversarial Networks,” *PMLR*, pp. 157–166, jul 2017.

[7] M. Zhang, Y. Zhang, L. Zhang, C. Liu, and S. Khurshid, “DeepRoad: GAN-based Metamorphic Autonomous Driving System Testing,” *CoRR*, vol. abs/1802.0, feb 2018.

[8] D. P. Kingma and M. Welling, “Auto-Encoding Variational Bayes,” *arXiv Prepr. arXiv1312.6114*, dec 2013.

[9] S. Kinoshita, T. Ogawa, and M. Haseyama, “LDA-based music recommendation with CF-based similar user selection,” *2015 IEEE 4th Glob. Conf. Consum. Electron. GCCE 2015*, pp. 215–216, jun 2016.

[10] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and É. Duchesnay, “Scikit-learn: Machine Learning in Python,” *J. Mach. Learn. Res.*, vol. 12, no. Oct, pp. 2825–2830, 2011.

[11] T. Karras, T. Aila, S. Laine, and J. Lehtinen, “PROGRESSIVE GROWING OF GANS FOR IMPROVED QUALITY, STABILITY, AND VARIATION,” tech. rep.

[12] Z. Hu, Z. Yang, R. Salakhutdinov, and E. P. Xing, “On Unifying Deep Generative Models,”

[13] T. Salimans, I. Goodfellow, W. Zaremba, V. Cheung, A. Radford, X. Chen, and X. Chen, “Improved Techniques for Training GANs,” in *Adv. Neural Inf. Process. Syst. 29* (D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, eds.), pp. 2234–2242, Curran Associates, Inc., 2016.

[14] K. Sohn, X. Yan, and H. Lee, “Learning Structured Output Representation using Deep Conditional Generative Models,”

[15] M. Lučić, K. Kurach, M. Michalski, S. Gelly, and O. Bousquet, “Are GANs Created Equal? A Large-Scale Study,” in *Adv. Neural Inf. Process. Syst.*, 2018.

[16] M. Mirza and S. Osindero, “Conditional Generative Adversarial Nets,” nov 2014.

[17] X. Mao, Q. Li, H. Xie, R. Y. Lau, Z. Wang, and S. P. Smolley, “Least Squares Generative Adversarial Networks,” in *2017 IEEE Int. Conf. Comput. Vis.*, pp. 2813–2821, IEEE, oct 2017.

[18] H. Lohninger, *Teach/me data analysis*. Springer, 1999.

[19] K. Hauser and Y. Zhou, “Asymptotically Optimal Planning by Feasible Kinodynamic Planning in State-Cost Space,” tech. rep.