Specification-Aided Trajectory Prediction with RNNs

Saarland University

Department of Computer Science

Master's Thesis

submitted by

Carsten Gerstacker

Saarbrücken, March 2020

Supervisor:	Prof. Bernd Finkbeiner, Ph. D.
Advisor:	Maximilian Schwenger
Reviewer:	Prof. Bernd Finkbeiner, Ph. D. Dr. Daniel Neider
Submission:	03 March, 2020

Abstract

Simulations of cyber-physical systems are used to avoid cost-intensive physical experiments. Even though the overall structure of the system might be well-studied, the exact dynamics can depend on undetermined parameters. Thus, modeling the system manually leads to additional expenses. Instead, the dynamics can be learned from existing trajectories.

Learning and predicting the highly non-linear dynamics of the full system, however, requires an expressive model such as recurrent neural networks and large amounts of training data. Expert knowledge of the system should be incorporated into the learning process to improve the learning rate and the quality of the model. Even though neural networks are in general considered to be black-box systems, feature preprocessing provides a convenient way to introduce additional knowledge into the learned model. We study the use of runtime monitoring specifications to calculate such features.

In this thesis, we aim to model cyber-physical systems with recurrent neural networks. We study the integration of RTLola specifications as a means to introduce expert knowledge into the model and aid the learning process. We evaluate our approach on a trajectory prediction task for quadcopters.

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement in Lieu of an Oath

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken, 03 March, 2020

Contents

1. Introduction 1

2. Preliminaries 3
- 2.1. Neural Networks . . . 3
  - 2.1.1. Training . . . 5
  - 2.1.2. Sequence Loss . . . 7
  - 2.1.3. Adam Optimizer . . . 7
  - 2.1.4. Curriculum Learning . . . 8
- 2.2. Recurrent Neural Networks . . . 8
  - 2.2.1. LSTM . . . 9
  - 2.2.2. GRU . . . 10
  - 2.2.3. Back-Propagation Through Time . . . 10
- 2.3. Regularization . . . 10
  - 2.3.1. Weight Decay . . . 11
  - 2.3.2. Dropout . . . 11
- 2.4. RTLola . . . 12

3. Trajectory Prediction 13
- 3.1. Data Generation . . . 13
  - 3.1.1. AirSim . . . 13
  - 3.1.2. Random Flights . . . 15
- 3.2. Preprocessing . . . 17
  - 3.2.1. Normalization . . . 17
  - 3.2.2. Displacement Vectors . . . 18
  - 3.2.3. Data Splitting . . . 19

3.3. Neural Network Architectures 19
3.3.1. Buffered Neural Network 20
3.3.2. Recurrent Neural Network 21
3.3.3. RTLola Integration 23

4. Evaluation **27**
4.1. Implementation 27
4.2. Models 28
4.3. Results 30

5. Conclusion & Further Work **35**

A. Code **41**

Chapter 1

Introduction

In domains like reinforcement learning, controllers of cyber-physical systems are trained to achieve a goal while avoiding critical failures. Even though the controller improves during learning, the training process is started with an arbitrarily bad controller. During these first training phases the controller randomly explores the environment in an trial-and-error manner, thereby gradually learning which actions yield good results and which bad results. Such training processes come with various failures and crashes until a sufficiently good controller is learned. Thus, training on real hardware involves high hardware costs. To avoid these cost-intensive physical experiments, simulations of cyber-physical systems can be used. Instead of repairing the physical object, failures can be recovered simply by restarting the simulation. Constructing good simulations, however, is a hard task. To simulate a dynamic system, the correspondence between control inputs and behavior of the system needs to be studied. This includes the system's reaction to the control inputs and the following physical behavior. Even though the overall structure of a system is known, the exact dynamics can additionally depend on undetermined parameters. For example, quadcopters come in various shapes and forms. Their dynamics are well-studied in general, but each drone depends on distinct constants. These have to be determined in physical experiments, thus leading to additional expenses whenever the system changes even slightly.

Instead of studying the dynamics manually, they can be learned from existing trajectories [1, 2, 3, 4]. The required trajectories can be produced in an safe environment, e.g., when a quadcopter is being controlled by an professional pilot. Thus, the control inputs and system behavior can be recorded without the system crashing. These trajectories can then be used to infer the dynamics of the system with machine learning, e.g., training a neural network to predict the systems behavior based on the control inputs. We are interested in simulating the system over multiple time steps, referred to as multi-step trajectory prediction. Learning the system dynamics over multiple time steps requires expressive models such as recurrent neural networks. To improve the learning process and the quality of the predictions, expert knowledge can be incorporated into

the learning process [5]. For example, when a quadcopter has a sensor to measure acceleration, the velocity of the quadcopter can be computed by integrating over the sensor value. This connection between acceleration and velocity is known and easy to compute. However, for a neural network it can be easier to use a precomputed velocity value instead of figuring out the connection between the sensor value and the dynamic behavior by itself. Due to neural networks being considered as black-box systems, it is hard to add information into the system. However, feature preprocessing provides a convenient way to introduce additional knowledge to the learning model. We propose the use of runtime monitoring specifications to calculate such features.

Runtime monitoring tools like RTLola [6] are used to supervise cyber-physical systems. Observing the system's behavior over time, the behavior is checked against a specification that ensures correct and safe behavior. The specification language in the case of RTLola is stream-based. Thus, the inputs are streams of data and specifications can express mathematical operations on stream values as well as aggregations over time. The specification language is powerful enough to express the calculation of useful features that can be used during training. Another advantage is the possible use in settings were RTLola specifications are already used. In that case, existing specifications can simply be reused for the training and might already hold valuable information for the network without further effort.

In this thesis, we model a cyber-physical system with recurrent neural networks. We integrate RTLola specifications as a means to introduce expert knowledge into the model and aid the learning process. We evaluate our approach on a trajectory prediction task for quadcopters.

Related Work Recurrent neural networks have been shown to be effective for multi-step prediction problems [7]. In general, multi-step predictions are hard because of the iterative predictions. Reusing erroneous predictions as a basis for successive predictions leads to an accumulating error over time [8]. The problem of modelling dynamic systems based on observed input/output pairs is also referred to as *non-linear system identification* [1]. Multiple architectures, including recurrent neural networks, have been proposed for the modelling task [2, 3, 4].

In addition to using neural networks to approximate a dynamic system, we aim to introduce additional knowledge into the system. Hybrid models (or Gray-Box models) [5, 9] inspire this idea as they embed computations specific to the training task into the network. In their work, quadcopter dynamics are integrated into the network directly. Note that our approach works similarly, but instead of integrating a fixed set of dynamics, we integrate a computation unit for features based on a specification. The specification can be changed without touching the model, and thus, our approach is more adaptable. Another difference comes from the fact that computation of gradients is inherent to neural network training. Thus, the computations that can be used inside of a network are limited to differentiable operations. Since we integrate RTLola as a preprocessing component, no gradients need to be computed for the additional input and the computation is free from limitations such as differentiability.

Chapter 2

Preliminaries

In this chapter, we introduce the background knowledge about the neural networks architectures used in this thesis and their successful training.

The *length* or *norm* of a vector $x = (x_0, x_1, \ldots, x_{n-1}) \in \mathbb{R}^n$ is defined as $|x| = \sqrt{\sum_{i=0}^{n-1} x_i^2}$. We notate the *dimension* of a vector $x \in \mathbb{R}^n$ as $\dim(x) = n$. *Concatenation* of two vectors $x \in \mathbb{R}^n$ and $y \in \mathbb{R}^m$ is defined as $x \oplus y = [x, y] = (x_0, x_1, \ldots, x_{n-1}, y_0, y_1, \ldots, y_{n-1}) \in \mathbb{R}^{n+m}$.

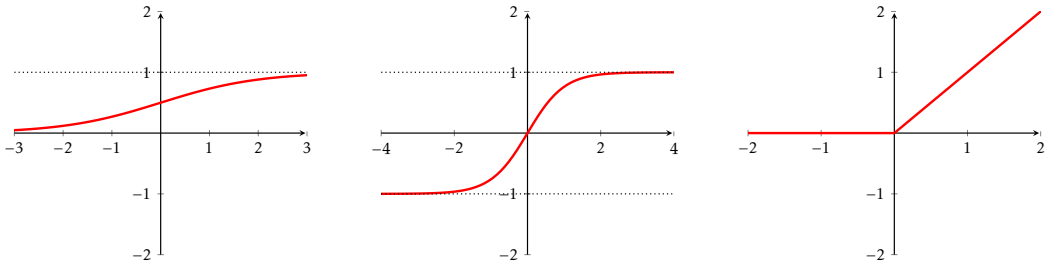
2.1. Neural Networks

Neural networks are structured computational units that are used to approximate arbitrary functions. The functions we want to approximate are in general unknown and only some samples are available. This samples are input-output pairs that correspond to the functions evaluation on a given input. The samples are referred to as *training data*. This data is used to fit the networks computation to match the unknown function. The networks computation depends on internal *trainable weights*. *Training* refers to the process of adjusting the trainable weights of the network such that the unknown function is approximated.

For example, a classic problem where neural networks are used is image classification. That is to predict the label of a given image, i.e., what is depicted on the image. The training data is a huge collection of pictures and the corresponding labels, e.g, pictures of dogs with the corresponding dog race as the label. The correct function to map images to labels is unknown, but a neural network is trained to approximate this function.

In this section, we formally introduce neural networks and explain the training process.

In the machine learning setting [10], a *neuron* refers to an operational unit that computes an weighted sum over its inputs $x \in R^n$. The sum is then passed into an *activation function* to make the computation more expressive. Activation functions refer to non-linear step-functions as depicted in Figure 2.1. For neural networks, the three



(a) Sigmoid σ **(b)** Hyperbolic Tangent *tanh* **(c)** Rectified Linear Unit (ReLU)

Figure 2.1.: The three activation functions mainly used in neural networks.

typically used activation functions are

$$\sigma(x) = \frac{1}{1+e^{-x}}$$

$$tanh(x) = \frac{sinh(x)}{cosh(x)} = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

$$\mathrm{ReLU}(x) = max(0, x).$$

All of the functions are differentiable with a non-zero gradient. This will later play a key-role in training the neurons to approximate arbitrary functions. Note that ReLU technically has no gradient at $x = 0$, but in practice the gradient is simply fixed to be 0. Let $w \in \mathbb{R}^n$ be the weights for each input and $b \in \mathbb{R}$, the function computed by a single neuron is

$$\sigma\left(x^\top w + b\right).$$

We refer to w and b as *trainable weights*.

A *neural network* is an expressive computational model consisting of layer-wise arranged neurons. The output of each layer, i.e., the output of each neuron in the layer, makes up the input to each neuron of the next layer. Layer-wise arranged neurons with individually simplistic behavior add up computational power and are able to compute complex functions. The goal is to adjust the trainable weights inside the neurons to approximate arbitrary functions. Note that without the activation functions, that introduce non-linearity into the individual units, the network would simply compute a polynomial.

We use neural networks to approximate a (possibly unknown) function merely based on samples from the original function. The difference between the network and the target function is measured with a *loss function* L. A common loss function is the *squared error* (SE)

$$L^{SE}(p, \hat{p}) = (p - \hat{p})^\top (p - \hat{p}),$$

where $p \in \mathbb{R}^n$ is the output of the approximated function and $\hat{p} \in \mathbb{R}^n$ is the network prediction. *Training* refers to finding weights w such that the function computed by the network f_w matches the unknown target function on all samples. The available

samples are referred to as *training data* and consist of tuples $(c_0, p_0), (c_1, p_1), \ldots, (c_d, p_d)$, where $c \in \mathbb{R}^{dim_{in}}$ is the input data and $p \in \mathbb{R}^{dim_{out}}$ is the prediction or output of the target function. Training is inherently an optimization problem with the goal to minimize the average loss over all possible training weights w

$$\min_w \frac{1}{d} \sum_{i=0}^{d} L^{SE}\big(p_i, \hat{p}_i\big)$$

$$\Leftrightarrow \min_w \frac{1}{d} \sum_{i=0}^{d} L^{SE}\big(p_i, f_w(c_i)\big).$$

2.1.1. Training

Since neural networks have lots of parameters, determination of the perfect weights to approximate a target function is computationally infeasible. Instead, one adjusts the weights in an iterative process to minimize the loss function. As mentioned earlier, the function computed by a neural network is differentiable. Thus, we can use *gradient descent* to follow the local gradient of the current weights to decrease the loss function L. Gradient descent is an optimization algorithm based on an iteratively applied optimization step

$$w_{t+1} = w_t - \alpha \frac{\partial L}{\partial w} \Big|_{w_t}$$

where $\alpha \in \mathbb{R}$ is the step-size. If w is a vector in $\mathbb{R}^n$, we use the gradient $\nabla_w L$ instead of partial derivatives $\frac{\partial L}{\partial w}$. Intuitively, the gradient is the local information about the direction in which the function decreases. The weights are adjusted in accordance to the gradient, and thus, the function is minimized. To compute the gradients with respect to the current weights, we use a technique called *backpropagation*. Instead of deriving the gradients for each weight by hand, the process can be automated when the derivation of each operation is known. The gradient of a function can then be computed step-wise by utilizing the *chain rule*

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y} \frac{\partial y}{\partial x}.$$

Example: Training Step

Consider the function $f_w(x) = x \cdot w_1 + w_2$, where $w \in \mathbb{R}^2$ is a weight vector parameterizing the function. Similar to the weights of a neural network, we want to learn the weights w to approximate another function. For this example, we want to approximate an unknown function g with one available sample. The available data is the sample $(2, 5)$ which corresponds to the function evaluating to $y = 5$ on the input $x = 2$.

We start the training process with randomly initialized weights $w = (1, 2)$. The loss function we aim to minimize is the *squared error* $L(y, \hat{y}) = (y - \hat{y})^2$. During the so called

forward pass, we compute the intermediate steps and memorize the intermediate results of each operation.

$$L(y,\hat{y}) \;=\; L(y, f_w(x)) \;=\; \Big(5-(\overbrace{\underbrace{2\cdot 1}_{o_*}+2}^{o_+})\Big)^2 \;=\; \underbrace{\overbrace{\big(5-4\big)}^{o_-}{}^2}_{o_2} \;=\; 1$$

Conveniently, the intermediate results can be identified by the operator, because each operation occurs exactly once, e.g., $o_- = 5 - o_+$. The results are $o_* = 2$, $o_+ = 4$, $o_- = 1$ and $o_2 = 1$. Note that $L = o_2$ since it is the last operation.

To minimize the loss function we are interested in the gradient $\nabla_w L|_{w=(1,2)}$. For this example, we only compute the gradient for w_1. The gradient computation is referred to as backpropagation because we traverse the computational graph in opposite direction to the forward pass. The gradients are only computed locally and put together with the chain rule. Looking at the following equations, one sees that using the chain rule we can compute the gradient stepwise by backpropagation through the computational graph.

$$\begin{aligned}
\frac{\partial L}{\partial w_1} &= \frac{\partial o_2}{\partial w_1} \\
&= \frac{\partial o_2}{\partial o_-}\frac{\partial o_-}{\partial w_1} \\
&= \frac{\partial o_2}{\partial o_-}\frac{\partial o_-}{\partial o_+}\frac{\partial o_+}{\partial w_1} \\
&= \frac{\partial o_2}{\partial o_-}\frac{\partial o_-}{\partial o_+}\frac{\partial o_+}{\partial o_*}\frac{\partial o_*}{\partial w_1}
\end{aligned}$$

This way we only need to compute the derivatives locally.

$$\frac{\partial o_2}{\partial o_-} = \frac{\partial (o_-^2)}{\partial o_-} = 2 * o_- = 2 \qquad \frac{\partial o_-}{\partial o_+} = \frac{\partial (5-o_+)}{\partial o_+} = -o_+ = -4$$

$$\frac{\partial o_+}{\partial o_*} = \frac{\partial (o_* + w_2)}{\partial o_*} = o_* = 2 \qquad \frac{\partial o_*}{\partial w_1} = \frac{\partial (4\cdot w_1)}{\partial w_1} = 4$$

Putting everything together, we have $\frac{\partial L}{\partial w_1} = 2\cdot(-4)\cdot 2\cdot 4 = -64$. Repeating the same computation for w_2, we get $\frac{\partial L}{\partial w_2} = -16$. Performing a gradient descent step with step size $\alpha = 0.01$, we get

$$w_1' \leftarrow w_1 - \alpha\cdot(-64) = 1.64 \qquad w_2' \leftarrow w_2 - \alpha\cdot(-16) = 2.16$$

and the updated function $f_{w'}(2) = 5.44$ is now closer to the approximated function. This steps are repeated multiple times and make up the training process.

2.1.2. Sequence Loss

Loss functions measure the error of the network predictions, i.e., they compare the prediction with the real data and compute the difference as a non-negative function. In our trajectory prediction setting, we compute the loss over sequences. A sample consists of an sequence of inputs $C = (c_0, c_1, \dots, c_n)$ and outputs $P = (p_0, p_1, \dots, p_n)$, where $c_t \in \mathbb{R}^{dim_{in}}$ and $p_t \in \mathbb{R}^{dim_{out}}$ are the inputs sampled in the t-th time step. The network is fed one input c_t at a time and tries to predict the corresponding output p_t. We denote to predictions of the network as $\hat{P} = (\hat{p}_0, \hat{p}_1, \dots, \hat{p}_n)$.

We use the *Sum-of-Squares-Error* (SSE) to measure the error between predictions and data. Let $n \in N$ be the sequence length and $p_t, \hat{p}_t \in \mathbb{R}^{dim_{out}}$ are sample data and prediction, respectively. The loss is then defined as

$$L^{SSE}(P, \hat{P}) = \sum_{t=0}^{n} (p_t - \hat{p}_t)^\top (p_t - \hat{p}_t).$$

We compute the averaged loss L_w over a set consisting of multiple sequences $D = \left((C^{(0)}, P^{(0)}), (C^{(1)}, P^{(1)}), \dots, (C^{(d)}, P^{(d)})\right)$ with respect to the current weights w as

$$\begin{aligned} L_w(D) &= \frac{1}{d} \sum_{i=0}^{d} L^{SSE}\left(P^{(i)}, \hat{P}^{(i)}\right) \\ &= \frac{1}{d} \sum_{i=0}^{d} L^{SSE}\left(P^{(i)}, f_w(C^{(i)})\right) \\ &= \frac{1}{d} \sum_{i=0}^{d} \sum_{t=0}^{n} \left(p_t^{(i)} - f_w(c_t^{(i)})\right)^\top \left(p_t^{(i)} - f_w(c_t^{(i)})\right). \end{aligned}$$

2.1.3. Adam Optimizer

The introduced gradient descent algorithm is the underlying idea of many optimization algorithms. In general, the optimization of the loss function is a hard problem and among many optimization algorithms, there is no superior algorithm. In an empirical comparison [11], *adaptive learning* algorithms were shown to be less sensitive to hyperparameters like the step-size and thus preferred.

Adam [12] is one optimizer in the class of adaptive learning algorithms. The algorithm uses exponentially decaying averages of the gradient and squared gradient to adapt the learning rate. The average over the gradients and squared gradients in each time step are referred to as *first moment estimate* and *second raw moment estimate* , respectively. Motivated by the physics of an rolling object, momentum refers to gradual changes to the gradient instead of abrupt changes. Imagining the gradient descent algorithm as a ball that rolls down on the function surface, momentum allows it to roll over flat local minima instead of stopping immediately. We use the standard implementation of the algorithm.

Gradient Clipping

In each training step, the gradients are computed and the weights are updated. When optimizing complex functions like neural networks, the gradients can grow too big such that the weights change too drastically and advancements during training are lost again. This problem is in general referred to as *exploding gradients*. *Gradient clipping* [13] refers to cutting of gradients before they are applied to the weights. In our case, we clip the gradients element-wise to 1.0, such that

$$clip(x) = \begin{cases} 1.0 & \text{if } x > 1 \\ -1.0 & \text{if } x < -1 \\ x & \text{otherwise.} \end{cases}$$

2.1.4. Curriculum Learning

Curriculum learning [14] refers to learning strategies that split the learning task into different complexities. It is inspired by the way humans study new problems. First, a student starts by learning simple concepts of the problem and then moves on to more difficult variations. For example, instead of learning a language by reading complex texts, one starts with small sentences and simple grammatical concepts. The difficulty of the content is then gradually increased until the student becomes able to understand full texts and complex grammar.

The stochastic descent method described earlier is only able to find a local minimum of a function. This is due to the search only considering local information instead of global. Depending on the starting point of the search, the algorithm finds the next local minimum. Mechanism like *momentum* in the Adam optimizer are used to pass over flat local minimum. However, it can not be guaranteed to find the global minimum of a function. Due to the neural network expressing high-dimensional functions, merely a global minimum is reached. In this setting, curriculum learning can be seen as a minimum search in a smoothened function. The idea also has a mathematical foundation in optimization theory, namely *continuation methods* [15]. First, the smoothened function is minimized and the trained weights saved. The chance of finding a global minimum in a smoothened function is higher due to the absence of some of the local minima. When increasing the complexity of the task, we build on the learned weights. Thus, the search starts in a region of the function that is assumed to be close to the global minimum.

2.2. Recurrent Neural Networks

In sequence prediction tasks, not only the current input but also the information as a whole is important. For example, when translating a sentence each word itself does not hold much information. However, each word in context with the sentence read so far holds meaning. Only in combination with the information about the whole sequence, the meaning of a sentence can be inferred.

Recurrent Neural Networks (RNN) refer to neural networks that step-wise read sequences of data and keep information over multiple time steps. This is achieved with recurrent connections that carry information over to the next time step. We explain the architecture of two recurrent neural networks used in this thesis.

2.2.1. LSTM

Long Short-Term Memory (*LSTM*) [16] units were first studied in 1997. In this architecture, the network not only learns the importance of the information for the current time step, but also the temporal importance over multiple time steps. The decision about the data that needs to be stored becomes part of the learning task and the information flow is controlled by trainable weights. This is achieved by three different *gates*, namely *forget*, *input*, and *output* gate. Additionally, an *internal cell* state is available and used to transfer information to the following time steps. The gates are used to control the information flow around the cell state and the output of the network.

Let c_t and $\hat{p}_t$ denote input and activation of the unit in the t-th time step, respectively. Each gate depends on the current input c_t and the units output of the last state $\hat{p}_{t-1}$. The trainable weights are referred to as W_x and b_x, where $x \in \{\mathcal{F}, \mathcal{I}, \mathcal{O}, C\}$ denotes the corresponding gate or node. The three gates compute a σ function that produces values in the range $[0, 1]$. Thus, multiplication with a gate corresponds to keeping or loosing information depending on the gate output being closer to 1 or 0, respectively. The forget gate

$$\mathcal{F}_t = \sigma(W_{\mathcal{F}} \cdot [c_t, \hat{p}_{t-1}] + b_{\mathcal{F}})$$

controls the information that is to be forgotten from the cell state. It is multiplied with the cell state to remove information. The input gate

$$\mathcal{I}_t = \sigma(W_{\mathcal{I}} \cdot [c_t, \hat{p}_{t-1}] + b_{\mathcal{I}})$$

controls the information that is about to enter the cell state. A *candidate value* for the new cell state

$$\overline{C}_t = tanh(W_C \cdot [c_t, \hat{p}_{t-1}] + b_C)$$

is multiplied with the input gate and the resulting vector is added to the cell state. Overall, the cell state is directly affected by the forget and input gate

$$C_t = \mathcal{F}_t \cdot C_{t-1} + \mathcal{I}_t \cdot \overline{C}_t .$$

The output gate again computes a σ function to control the output of the unit

$$\mathcal{O}_t = \sigma(W_{\mathcal{O}} \cdot [c_t, \hat{p}_{t-1}] + b_{\mathcal{O}}).$$

Finally, the prediction of this unit is

$$\hat{p}_t = \mathcal{O}_t \cdot tanh(C_t).$$

Note that LSTM naturally has a large amount of trainable weights, due to the amount of different gates.

2.2.2. GRU

Gated Recurrent Units (*GRU*) [17] are an simpler version of LSTM units. First of all, there is no explicit cell state but its last output is reused. Furthermore, the forget and input gate are merged into one component, thus, reducing the amount of trainable weights. Again, we denote input and activation of the unit by c_t and $\hat{p}_t$, respectively. The two gates used in the GRU architecture are referred to as *reset gate* $\mathcal{R}_t$ and *update gate* $\mathcal{Z}_t$

$$\mathcal{R}_t = \sigma(W_{\mathcal{R}} \cdot [c_t, \hat{p}_{t-1}])$$
$$\mathcal{Z}_t = \sigma(W_{\mathcal{Z}} \cdot [c_t, \hat{p}_{t-1}]).$$

The reset gate is used to compute a new candidate activation $\bar{p}_t$

$$\bar{p}_t = tanh(W_h \cdot [x, \mathcal{R}_t \cdot \hat{p}_{t-1}])$$

by controlling the influence of the last activation. The *update gate* is used to interpolate between the last activation and candidate activation. Altogether, the activation is

$$\hat{p}_t = \mathcal{Z}_t \cdot \hat{p}_{t-1} + (1 - \mathcal{Z}_t) \cdot \bar{p}_t .$$

Note, that GRU units have significantly less trainable weights than LSTM units. Therefore, they can be better suited when less training data is available. However, in general LSTMs outperform GRU units.

In the literature, there are many other LSTM variations and their differences were studied in [18]. In general, there are no significant improvements over the classic LSTM architecture. We use the two presented architectures and compare their efficiency with respect to the available amount of data.

2.2.3. Back-Propagation Through Time

Due to the recurrent connections of RNN units, classic backpropagation algorithms can not be directly applied to recurrent architectures. An adjusted training algorithm for recurrent neural networks is referred to as *Back-Propagation-Through-Time* (*BPTT*) [19, 20]. To train a recurrent neural network, the recurrent connections are unrolled over the prediction time steps. Thus, multiple copies of the network are created and recurrent connections simply connect the copies of adjacent time steps. The forward pass is computed on the unrolled network. Then, the loss of each prediction step is computed and backpropagation is performed. The backpropagation starts in the last time step and propagates through the recurrent connections. This allows for traversing accumulated errors back through time – hence the name – and learn temporal dependencies.

2.3. Regularization

Overfitting is a problem that occurs when approximating sampled training data by overly complex functions. Due to neural networks being highly non-linear functions, they are

very prone to overfitting on training data. One way to detect overfitting is by observing the *generalization error* of a model. This refers to the error a model has on data it was not trained on. During training the error on the training data naturally decreases as well as the error on a separated valuation set. In the case of overfitting, however, the generalization error starts increasing while the training data continuously drops. Thus, the model is to precise on the training data, such that it does not generalize on the problem. Two commonly used methods to prevent overfitting are weight decay and dropout.

2.3.1. Weight Decay

Weight decay – also known as L^2 parameter regularization – penalizes the use of high magnitude weights in the model. The loss function is extended by an additional term that needs to be minimized together with the loss L_w

$$\min_w \; L_w + \lambda \cdot |w|^2$$

where w are the trainable weights and $\lambda \in \mathbb{R}$ is a *regularization parameter*. The regularization parameter λ controls the trade-off between loss and regularization. With an arbitrary big parameter, the loss of the optimization problem would be completely neglected and merely the weight penalty would decide. Since this is not the goal, the parameter is generally rather small, e.g., 10^{-3}, 10^{-4} or 10^{-5}.

For example, assume that two sets of weights w_1 and w_2 are able to achieve a similar loss $L_{w_1} \approx L_{w_2}$. For better generalization on the training task and to avoid overfitting, we prefer the simpler model. With weight decay, the complexity of a model is measured by the squared norm of the weights, thus big weights are penalized more. If $|w_1|^2 < |w_2|^2$, the weights w_1 would be preferred over w_2, because they represent a simpler model.

This allows us to use expressive models like neural networks, but within the possibly computed functions we settle for the smoother ones.

2.3.2. Dropout

Dropout [21] is a regularization method for neural networks. Dropout is a simple method to prevent overfitting during training and corresponds to averaging the output of multiple models. Instead of training a fixed network structure, in each training step randomly chosen neurons are deactivated. Deactivating or disabling a neuron simply corresponds to multiplying its output with 0. The probability of disabling a neuron $p_d \in [0,1]$ is a hyperparameter and is chosen prior to the training. Deactivating neurons randomly corresponds to training multiple *subnetworks*, i.e., the network consisting of the remaining active neurons. In general, there exist exponentially many subnetworks that can be created by randomly deactivating sets of neurons. It would be computationally infeasible to train all subnetworks explicitly and then average their outputs. Instead, dropout samples a subnetwork in each training step. During prediction, all neurons are activated and the weights are scaled down in accordance to the dropout hyperparameter. The prediction then approximates the expectation of multiple subnetworks.

2.4. RTLola

RTLola [6] is a stream-based monitoring specification language. Inputs are considered to be streams of data and the specification describes computations on these streams such as accumulation and integration. In RTLola, input streams can be asynchronous, i.e., the data of the input streams does not necessarily arrive at the same time. When working with cyber-physical systems, one can not assume data to arrive synchronously due to sensors and different components having individual sampling rates. In our case, multiple sensors of quadcopters are used, and thus, data is in general asynchronous.

Given a specification in the RTLola syntax, the input streams can be processed and output streams are calculated accordingly. The memory consumption of the monitoring tool can be computed based on a static analysis of the given specification. Due to the modular design of RTLola, computations can be highly parallelized and compiled onto FPGAs to be used efficiently in closed looped systems [22].

As an example of RTLola specification, consider two input streams that correspond to the current x and y velocity of an moving object. Assuming we want to track the position of the object, that is a stream for the x and y coordinates, respectively. The corresponding RTLola specification looks as follows.

```
input velocity_x: Float64
input velocity_y: Float64
output position_x@10Hz:= position_x.offset(by:-1).defaults(to:0.0) +
    velocity_x.aggregate(over:0.1s, using:integral)
output position_y@10Hz:= position_y.offset(by:-1).defaults(to:0.0) +
    velocity_y.aggregate(over:0.1s, using:integral)
```

First, the two input streams and their types are defined. The output streams in this example use multiple features of RTLola. The `@10Hz` annotation defines the update frequency, i.e., the stream needs to be updated 10 times a second. The integral over the velocity over the last 0.1 seconds is computed with `aggregate(over:0.1s, using:integral)`. To sum up the integrals over time, we access the last computed output of the stream with `offset(by:-1)` and pass a default value for the first step. Every update corresponds to the sum over fixed size integrals over the input stream.

Chapter 3

Trajectory Prediction

Predicting trajectories of dynamic systems is a *multi-step prediction problem*. Instead of only making a single prediction, the network will be used to predict a trajectory consisting of multiple steps. These steps are not estimated altogether but iteratively building up on each other. Given the current state and control inputs, the network is trained to predict the next state of the trajectory. By repeating this process and reusing predictions, arbitrary long trajectories can be generated.

However, due to the iterative nature of the prediction process the error of each step affects the future predictions. Since each step is based on the previous one, the error is accumulated over time. We refer to this inherent problem as the *accumulation error*.

In this chapter, we present our approach to predicting dynamic systems over multiple time steps. The concrete problem is the step-wise estimation of flight trajectories of quad-copters. The approach is logically separated into data generation and neural network architectures.

3.1. Data Generation

The data generation is based on *AirSim* [23], a simulation tool developed by Microsoft. We use the tool to simulate quad-copters in real-time and track their dynamic behavior. The recorded flights will later be used as training data for the neural networks.

In this section, we first introduce the AirSim tool and the generation of random flights.

3.1.1. AirSim

AirSim is based on *UnrealEngine* [24] to simulate quad-copters and cars in real-time. UnrealEngine is a 3D game engine used for the game development and real-time simulations. Additionally, AirSim offers a python module to interact with the simulation. We use the interface to control the simulated quad-copters and track the dynamic behavior.

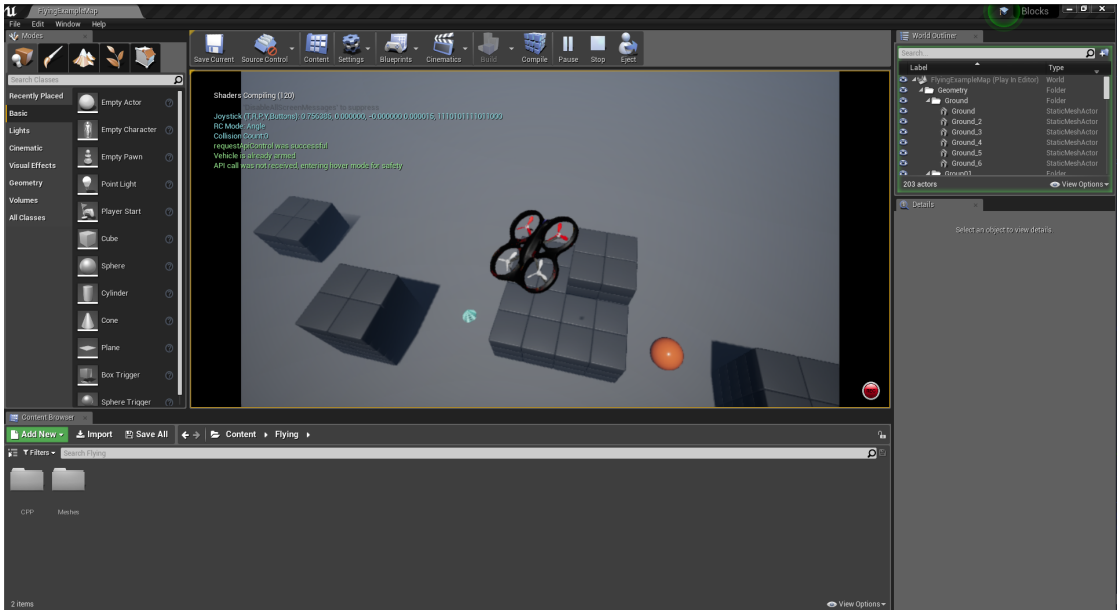


Figure 3.1.: A screenshot of an AirSim simulation running in UnrealEngine. The quad-copter is randomly controlled and the dynamic behavior recorded.

Figure 3.1 depicts the UnrealEngine during simulation of an AirSim quad-copter. The environment is the standard AirSim environment "Blocks4.18".

The drone is controlled by four signals [25], namely the *pitch*, *roll*, *throttle* and *yaw rate*. Throttle is used to adjust the overall thrust the four rotors generate. Depending on the throttle, the quad-copter either flies upwards or falls down. The remaining three signals are used to control the drone's movement and orientation. Each signal corresponds to a axis the drone is rotated around, e.g., pitch and roll tilt the drone forward/backwards and left/right, respectively. Yaw rate corresponds to ration around the relative z-axis, i.e., whether the drone rotates clockwise/counter-clockwise when looked at from above. Throughout the remaining thesis we will denote these four control signals as $c = (pitch, roll, throttle, yaw)^\top$.

The state of the quad-copter is measured by multiple sensors. An *IMU* unit measures the orientation, angular velocity and linear acceleration. The IMU unit provides the arguably most important measurements for the prediction task. Both velocity and acceleration are useful for the prediction of dynamic behavior. Another available sensor is a *barometer* that provides altitude and pressure measurements. These can be used to access the current height of the quad-copter. The remaining sensors available are a *magnetometer* and a *gps* unit.

Since the environment is simulated, we can also access the vehicles *ground-truth kinematics*. These are the position and acceleration data used within the simulation to

calculate the system dynamics. The positional values are given as absolute positions, that is the exact position w.r.t to the simulations coordinate system.

The ground-truth kinematics together with the sensor values make up the state information of a single time step. We refer to this information as state and position, interchangeably. We denote recorded positions from the simulation by p and position predictions from neural networks by $\hat{p}$.

3.1.2. Random Flights

The remaining piece for the data generation is the control input. With AirSim taking care of the simulation, there are two ways to control the quad-copter. The control signals can be generated by either manual labor, i.e., a human manually controlling the quad-copter, or an automated controller. Given the amount of data required for neural network training, we utilize a controller to randomly generate the control signals. The generated signals are send to AirSim in real-time and the quad-copter behavior is recorded. This now leaves two tasks to be taken care off, namely the random signal generation and the quad-copter state sampling. We start with latter problem.

Sampling

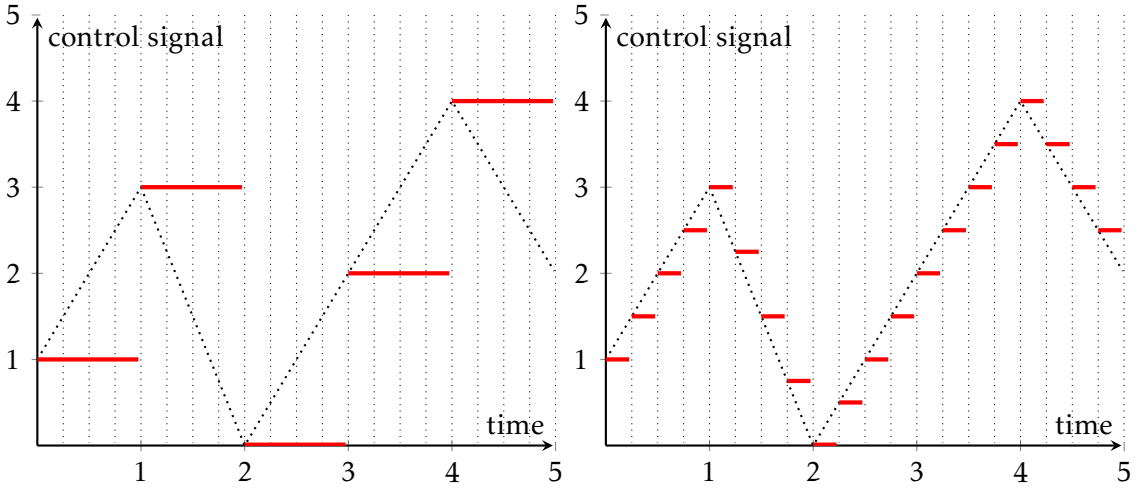
Sampling in the context of signal processing is the conversion of a continuous signal to a discrete one. For discrete signals, we refer to the amount of steps per second as the *sample rate*.

The problem we are facing is to sample the quad-copter dynamics into a discrete signal that can be stored. Note that the simulation of the quad-copter is technically already a discrete signal because it is computed step-wise. However, the sample rate of the simulation is much higher than the amount of data we can store. Instead of recording each simulation-step, we sample the signal with a lower sample rate and thereby reduce the memory usage.

Besides the memory required to record all simulation-steps, there is another reason to favour a lower sample rate. The sampled data will later be used to train a neural network. Thus, the network learns to make step-wise predictions in accordance to the sample rate. For example, if the training data is sampled once every second, the network learns to predict one second into the future. Considering the accumulation error introduced earlier, higher sample rates correspond to more predictions, and thus, a higher accumulation error.

On the other hand, choosing the sample rate too low also has some disadvantages. Given that the training data is simulated in real-time, the sample rate directly corresponds to the amount of training data that we are able to produce each second. Since neural networks are known to require lots of data to train, a reduced sample rate bares the risk of an insufficient amount of training data.

We use two sample rates and later compare the results. Each random flight will be simulated for ~ 30 seconds. We choose the sample rate to record at least $128/512$



(a) Piecewise Linear **(b)** Interpolated

Figure 3.2.: Different ways to generate the control inputs (red lines). In (a), the control signals are piecewise linear over ~ 1.5 second periods. During this time the dynamic system state is sampled multiple times as depicted by the dotted vertical lines. The control signals of (b) are interpolated in accordance with the sample rate to avoid disruptive changes in the input.

time steps per trajectory. Due to the interface with AirSim blocking at times, the communication with the simulation can not be perfectly timed. Thus, some small noise is introduced into the sampling process. This noise can cause the actual sample rate to be slightly smaller than anticipated. Considering this behavior, we choose the sample rates to be 5 Hz and 18 Hz, respectively. This way the anticipated amount of samples per trajectory is slightly exceeded and we can shorten the traces subsequently.

Random Controls

In accordance to the sampling methods, we propose two kinds of randomly generated controls. Together with the low frequency, we generate control inputs that are constant over a few sample steps. We refer to this kind of control inputs as *piecewise-linear*. The controls are randomly generated every $\sim 1-1,5$ seconds and then kept constant until the next controls are generated. The corresponding simulation is sampled multiple times over each interval as depicted in Figure 3.2a.

The second kind, as depicted in Figure 3.2b, changes the inputs between each sampling step. Instead of keeping the control signal constant over multiple sample step, two randomly chosen controls are linearly interpolated over a few sampling steps. This way each input is close to the next one and there are now disruptive changes in the control signal. This method is combined with the high-frequency sampling to smoothen the

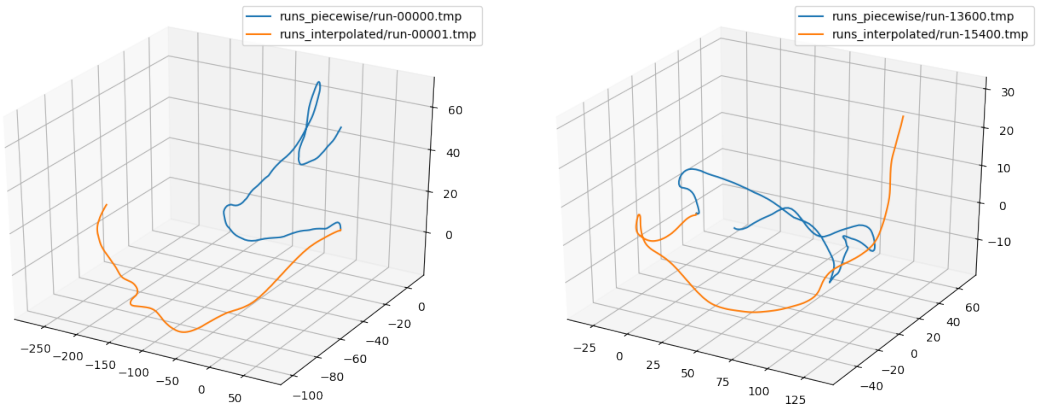


Figure 3.3.: Plot of random flight trajectories. The trajectories were simulated with piecewise-linear (blue) and interpolated (red) random controls. Compared to the trajectories from piecewise-linear controls, the interpolated controls produce slightly smoother trajectories.

interpolation. The corresponding trajectories will thereafter be referred to as *interpolated*.

With each sample we record the current control inputs c together with the state p. The trajectories then consist of a sequence of tuples $(c_0, p_0), (c_1, p_1), \ldots, (c_n, p_n)$, where c_i and p_i refer to the i-th control and i-th state sample, respectively. Figure 3.3 depicts two generated random flights. The plots depict only the three dimensional position information. The interpolated controls produce smoother trajectories in general, but also contain parts where the direction changes abruptly.

3.2. Preprocessing

Prior to training the neural network, we prepare the input data to make it more suitable for the training process. The data preparation is referred to as *preprocessing*. We apply two kinds of preprocessing steps to our data as explained in the following.

3.2.1. Normalization

Normalization is used to improve the performance of neural networks [26]. The idea is to transform all features in the training data to a fixed range. Therefore, all features are equally weighted, independent of their initial magnitude. For example, the position data sampled from AirSim is from the interval $[-100, 100]$. In contrast to this, the control signals are from a very small range $[-1, 1]$. Normalizing the data such that all features are in the same interval improves the performance of the network as well as speeding up the training process.

In this work we use *Min/Max normalization*. The goal is to scale the data to an interval between 0 and 1. We denote normalization as a function f on the training data d. In our

case, the data $D \in \mathbb{R}^{n \times l \times dim_{in}}$ is a three dimensional tensor. The dimensions correspond to the number of trajectories n in the training set, the length of each trajectory $l \in \{128, 512\}$, and the feature dimension dim_{in}, respectively. The feature dimension is the amount of inputs the network receives each prediction step. In our case, this corresponds to the concatenation of the control signals $|c| = 4$ and state signals $|p| = 10$, thus $dim_{in} = 14$. We calculate the minimum $min(D) \in \mathbb{R}^{dim_{in}}$ and maximum $max(D) \in \mathbb{R}^{dim_{in}}$ with respect to each feature. Normalization is then defined as

$$f(D) = \frac{D - min(D)}{max(D) - min(D)}.$$

The operations are broadcasted to be applied on each entry of d. Formally,

$$f(D)_{i,j,k} = \frac{D_{i,j,k} - min(D)_k}{max(D)_k - min(D)_k} \quad \forall i, j, k.$$

For example, assume the min and max of the *throttle* control signal to be 1.3 and 0.5, respectively. Then, every throttle signal in the training data t is mapped to $\frac{t-1.3}{0.5} \in [0, 1]$.

Denormalization

The network trains on normalized data and the network predicts normalized values. These predictions are reused to iteratively predict multiple steps as discussed earlier. Note that an additional normalization step is not required, because the predictions already correspond to normalized values. However, when using the network for inference, we expect the predictions of the network to be denormalized. To compute the final output, the operation inverse to normalization is applied on the outputs. We refer to the inverse operation as *denormalization* and denote it by f^{-1}.

3.2.2. Displacement Vectors

As mentioned in Section 3.1.1, the ground-truth positions are absolute positions with respect to the simulation origin. These positions are distributed over a big range of numbers and distinct to the simulation environment. For example, when predicting in an environment with a different origin, the network has to deal with values very different to the training data. Even though, the dynamic behavior of the quad-copter does not change depending on the origin, the networks will perform worse on these unseen inputs. Therefore, we introduce an additional preprocessing step to convert absolute positions to relative ones. We refer to the relative position between two time steps as *displacement vector*. The displacement vector is calculated by subtracting two subsequent time steps. Note that the magnitude of the displacement vectors is implicitly bounded by the maximal velocity of the quad-copter and independent of the simulation origin. Due to the bounded magnitude of the displacement vector, it is guaranteed that the positional values will stay familiar to the values encountered during training. When plotting a trajectory, the displacement vectors are accumulated to calculate the absolute position.

In contrast to absolute values, however, the displacement error does not hold the information about the trace as a whole. Therefore, the accumulation error over multiple steps is not present, and thus, cannot be used to guide the predictions. We empirically compare both position representations and their influence on the prediction quality in the evaluation part.

3.2.3. Data Splitting

The preprocessed training data is split in three parts, namely training, validation and test data. The training data, making up 60% of the overall data, is actively used to train the network. Validation data makes up 30% of the data and is not used to train the network, but to find suitable hyperparameters. The loss with respect to the validation data is computed once every training epoch. This validation loss is used to measure the networks performance on data it was not trained on. It indicates the generalisation performance and possible overfitting during the training process. The remaining test data is used as performance assessment at the end of the training. In contrast to training and validation data that are respectively used for weight and hyperparameter search, the test data has no direct influence on the network.

3.3. Neural Network Architectures

In this section, we introduce three different neural network architectures. For multi-step trajectory prediction, we need to predict sequences of data. Information that can be inferred from multiple time steps will be called *temporal information*. For example, to predict a moving dynamical system the acceleration is a necessary feature, because it describes the change between two system states. If the acceleration is not given directly, it can only be inferred over multiple time steps. Due to the importance of temporal information to dynamic system prediction, we focus on architectures able to access temporal information.

We start with buffered feed-forward networks which will make up the baseline against which the other approaches are compared. The buffer stores a fixed amount of inputs. This gives the network access to not just the current but also past input data. The next model are recurrent neural networks that internally keep track of data over multiple time steps. In contrast to buffers that only give a limited static view into the past, recurrent neural networks actively learn which data to store. Thus, allowing the network to keep information over arbitrary many time steps. In an attempt to increase the prediction quality, we integrate RTLola as preprocessing engine to recurrent neural networks. RTLola maintains an internal state and processes data in accordance to custom specifications. This allows to explicitly extract and process temporal information during training and prediction.

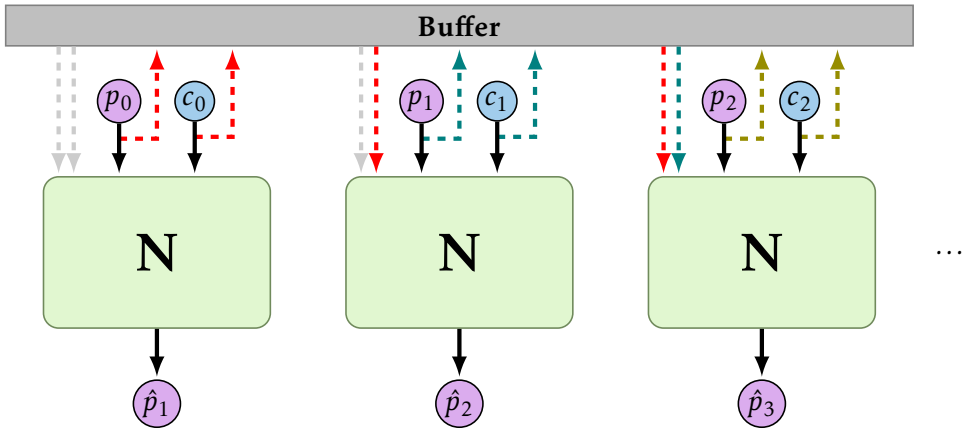


Figure 3.4.: A buffered feed-forward network. The inputs are stored in a buffer of size 2 and can be accessed by later time steps. Dashed lines of same color depict data of the same time step. During the first steps the buffer defaults to 0, depicted by gray arrows.

All networks aim to predict the next state of the dynamic system, namely the sensor states and position. The input to the network at time step i is the current state c_i and control inputs p_i.

$$input_i = p_i \oplus c_i\,.$$

During inference, multiple time steps are predicted by iteratively reusing the predicted states as input to the network.

$$input_i = \hat{p}_i \oplus c_i\,,$$

where $\hat{p}_i$ is the prediction for the i-th step, i.e., the prediction made in step $i-1$.

3.3.1. Buffered Neural Network

We use *buffered neural networks* as the baseline to compare the more advanced models against. To access events of the past, both positional data and control inputs are saved in a fixed-size buffer. As depicted in Figure 3.4, the inputs are forwarded to the buffer and can be accessed in future time steps. The dashed lines depict information stored into and read from the buffer. Lines of same color belong to the same time step, e.g., the red lines correspond to the first time step and can be accessed during the second and third time step. However, the data of the first time step can not be accessed during the forth or later steps, because the size of the depicted buffer is 2. The buffer is initiated with 0s, depicted as gray lines. Thus, the network behaves correctly after the first two steps when the buffer is filled. These first few steps make up the *initialization phase*.

In contrast to NARX [27, 28] – a class of recurrent neural networks with additional memory – we only buffer the inputs and do not forward the gradients through the buffer during training.

A buffer of size k only stores information about the last k time steps. Formally, the input of time step i including the buffer information is

$$input_i = \left(p_{i-k} \oplus c_{i-k} \oplus \ldots \oplus p_{i-1} \oplus c_{i-1}\right) \oplus p_i \oplus c_i \,,$$

where $k \in \mathbb{N}$ is the buffer size and the parentheses represent the buffer. During inference, all states p_j are replaced by the network prediction of the last time steps $\hat{p}_j$.

A disadvantage of this structure is the limited access to past information. Only the last k time steps can be kept in the buffer at a time. Additionally, the network has no control over the stored data besides using them as input. We refer to this kind of buffer as *static memory*, because of the data simply being stored without any calculation applied.

Buffer Size

The buffer size can be thought of as an additional hyperparameter. It introduces a horizon for past events such that any kind of information outside of it can not be accessed anymore. Thus, the size needs to be chosen in accordance to the problem. In general, choosing a sufficient buffer size is not easy because it might be unclear which information of the past is necessary.

In our case, the network needs to predict the movement of a dynamic system. As argued earlier, an important information is the velocity as well as the acceleration of the system. When using the displacement vectors, we already have a linear approximation of velocity with respect to the length of each time step. An additional time step allows the comparison of the past and current velocity, i.e., an linear approximation for the current acceleration. If we naively assume that this information is sufficient, only a single time step is required for the prediction task. However, this argument is merely based on the positional information and does not consider any of the other state information, let alone the required buffer size to accurately predict them. Additionally, the buffer size corresponds to an increase in the networks input dimension. Therefore, the overall network size and structure might be in need of adjustments. We decide to evaluate three different buffer sizes 2, 4 and 6 and evaluate their performance empirically.

3.3.2. Recurrent Neural Network

In contrast to an input buffer, recurrent neural networks have an internal memory. The information is kept in the internal cell state of fixed size and consists of floating-point numbers. Trainable weights regulate the information flow through the cell state. Thus, decisions about whether to keep or remove information becomes part of the training problem. In contrast to static memory methods like buffers, the network decides the importance of individual events and influences the memory behavior. Information that is not considered to be useful can be dropped, thus freeing memory to store other information.

An advantage of this structure is the possibility to keep information over long periods of time. Since there is no implicit event horizon with respect to time steps, older steps

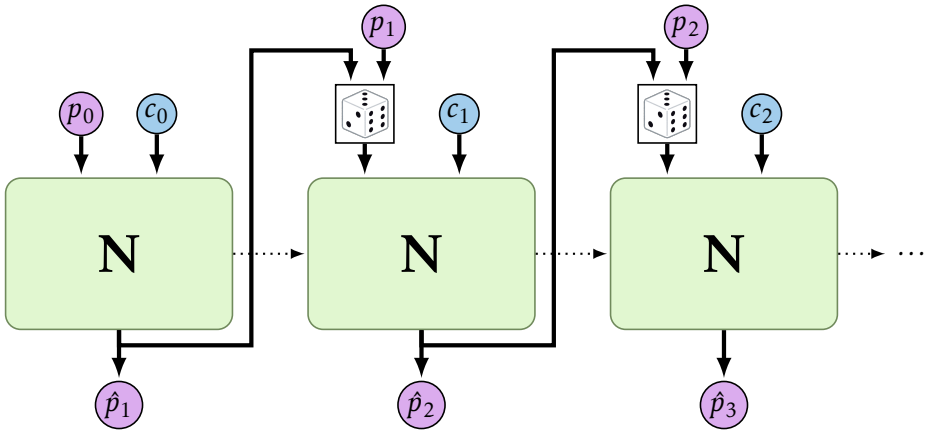


Figure 3.5.: A recurrent neural network with scheduled learning strategy. The network has an internal state to store information over multiple time steps. This recurrent connections are depicted by dotted lines. According to scheduled sampling, the choice to reuse the prediction or use the training data is randomised.

are not automatically overwritten. Instead, the information can be stored over arbitrarily long time periods. Learning long term dependencies is a hard problem in general, but recurrent neural networks – especially LSTMs – are known for their good performance [29]. Additionally, there is no need for extra hyperparameters that require manual fine-tuning for each individual problem.

The generalized information flow of recurrent neural networks is depicted in Figure 3.5. The dotted lines represent the internal state of the network being passed to the next time steps. For what follows dotted lines are used to depict temporal connections between time steps. This can refer to recurrent connections and internal states of components, e.g., static memory.

As introduced in Section 2.2, we use two kinds of recurrent neural network architectures, namely GRU and LSTM. In the following sections, we refer to GRU and LSTM simply as recurrent neural networks. Both variants will be compared in the evaluation chapter.

Scheduled Sampling

According to Section 2.1.4, we refer to training strategies that gradually increase the problem complexity during training as *curriculum-learning* strategies. For recurrent neural networks, we implement an improved version of *scheduled sampling* [30]. During training the network typically predicts trajectories exclusively based on the training data as inputs. The predicted trajectory is only used to compute the loss function and the gradients. This behavior varies significantly from the inference behavior where the network is forced to iteratively reuse prior predictions. Thus, the network is confronted

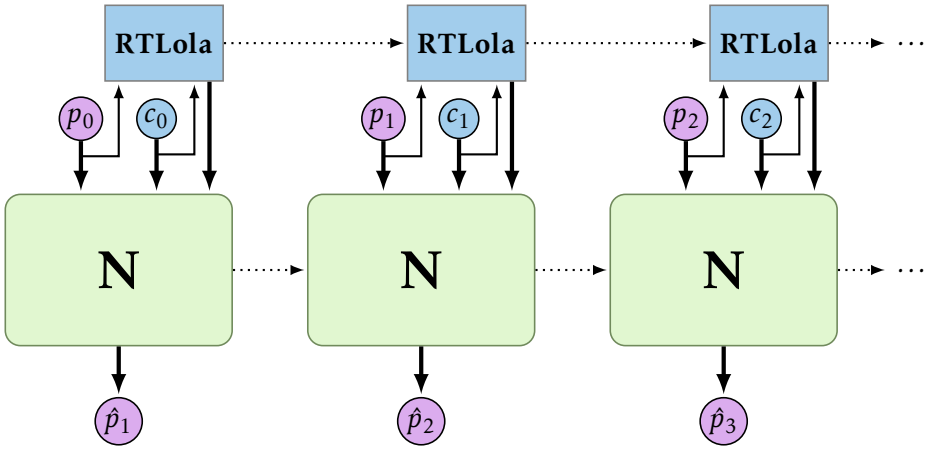


Figure 3.6.: Recurrent neural network with RTLola integration. The inputs are forwarded to RTLola to compute additional features. The internal states over time as well as recurrent connections are depicted by dotted arrows.

with unfamiliar values and small errors in each time step can easily accumulate over time.

Scheduled sampling aims to resolve this gap by bringing the behavior during training and inference closer together. During the training process, training data is gradually being replaced by prior predictions. Thus, the network is forced to behave similar to the inference task. The choice to either use a training data position p or reuse the prediction $\hat{p}$ is decided randomly in each step. We refer to the probability ϵ to reuse the prediction $\hat{p}$ as *schedule parameter*.

Due to the arbitrarily bad performance during the first training iterations, reusing predictions all the time would slow down convergence of the training process. Instead of a constant probability, ϵ gradually increases along the training iterations. Thus, first learning from the training data and bit by bit increasing the probability to reuse predictions, and thus, resemble inference.

In the original implementation, the gradients are not propagated through the predictions of earlier steps and the propagation is explicitly mentioned as further work. We implement the scheduled sampling algorithm in TensorFlow. By utilizing the automated gradient computation and appropriate structures, the gradients are propagated through the reused predictions. However, we do not compare the difference in the two approaches. The implementation is given in Algorithm A.3.

3.3.3. RTLola Integration

In this section, we extend recurrent neural networks with RTLola. So far we have introduced two different methods to handle temporal information. First, buffered neural networks with fixed-size static memory were presented. Then, recurrent neural networks that use internal memory which is controlled by trainable weights. Introducing

RTLola as a preprocessing unit can be compared to extending recurrent networks with additional memory and computational power. In addition to the network learning to handle temporal information, an expert describes the memory behavior by means of a specification. In contrast to using a simple static memory component, RTLola does not only introduce additional memory, but describes the extraction of temporal data. Thus, introducing expert knowledge that simplifies the training task, e.g., well-known physical laws.

For example, the quad-copters have an integrated IMU unit. As stated in Section 3.1, this provides us with an accelerometer and enables us to measure the acceleration of the drone. From the physics point of view, velocity can be derived by integrating over the acceleration. Even though, a recurrent neural network has the computational power to compute this integral, the precise computation first has to be learned. In RTLola, we can easily express such integrals and thereby simplify the learning problem for the network. Furthermore, the network can use the precomputed features instead of utilizing the internal memory. Thus, the internal memory requirements are relaxed.

Figure 3.6 depicts the extension of recurrent neural networks by RTLola. The inputs are forked to the RTLola component and then processed. The produced output of RTLola is then used as additional input to the network. Formally, the input of the i-th time step is

$$\bar{p}_i = \begin{cases} \hat{p}_i & \text{with probability } \epsilon \\ p_i & \text{otherwise.} \end{cases}$$

$$input_i = \bar{p}_i \oplus c_i \oplus RTLola_i(\bar{p}_i, c_i)$$

For inference, the network predictions are always reused, i.e., $\bar{p}_i = \hat{p}_i$.

Monitor Normalization

Our neural networks are trained on normalized data, i.e., inputs to the networks and outputs from the networks are within a normalized domain $[0, 1]$. However, the specifications for RTLola are written by hand and refer to the original values before normalization is applied. To integrate RTLola, we denormalize the inputs before forking them into the RTLola component. Especially during inference, where the position is only available in the normalized domain, incoming data to RTLola needs to be denormalized.

Note that normalization is also applied to the RTLola monitor output. The normalization data for the monitor is once computed in the beginning of the training. For this matter, the denormalized training data is used step-wise to compute RTLola outputs. This data is then used to compute the normalization parameters that are required to normalize the monitor outputs. The exact timing for normalization is depicted in Figure 3.7.

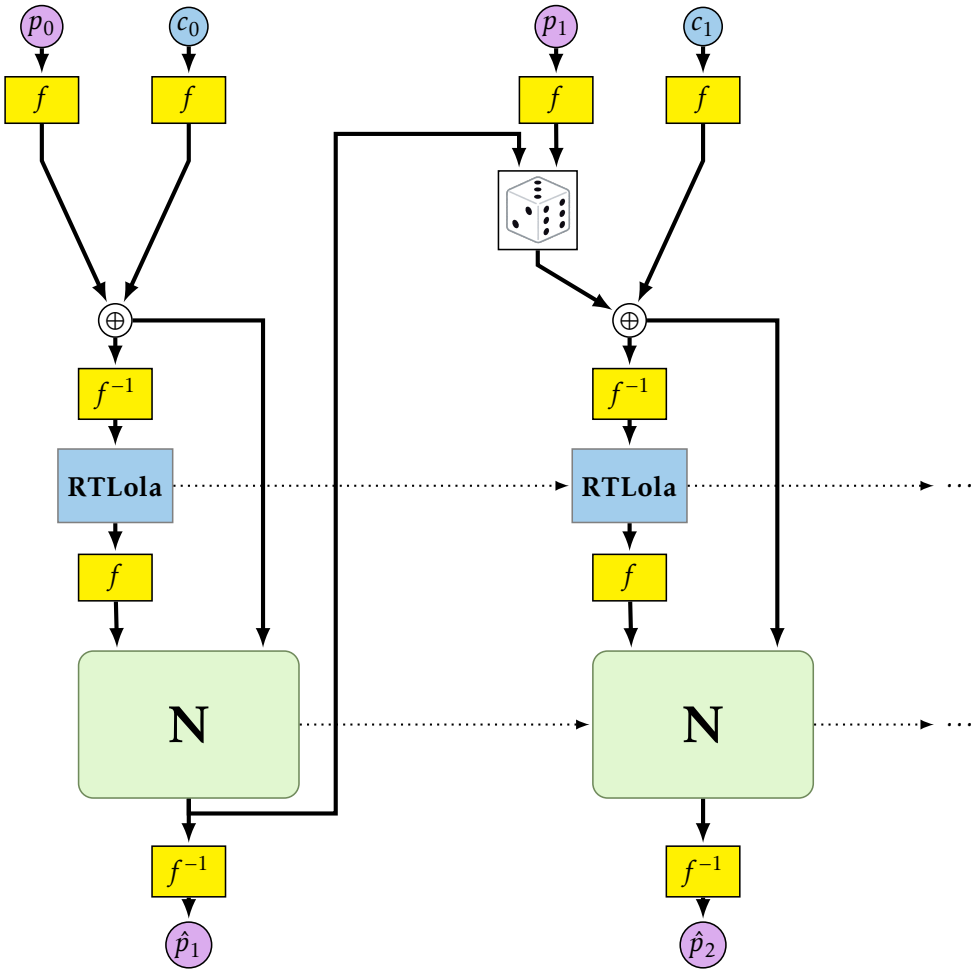


Figure 3.7.: Two time steps of a neural network with RTLola integration and curriculum learning. The yellow boxes with labels f and f^{-1} represent the normalization and the inverse operation, respectively. Because specifications for RTLola refer to the true values, denormalization is applied before passing values to the RTLola component. The output of the network is forwarded to the next step. In accordance to scheduled sampling, it is randomly chosen between the training data or prediction of the earlier step. The random choice is depicted by a dice and dotted lines represent internal states connections.

Runtime Trade-Off

The RTLola integration introduces a trade-off between computational speed and prediction quality. Training neural networks is usually done on GPUs, because the computations can be parallelized. Thus, the training can be highly accelerated and computational speed is increased. The calls to RTLola, however, happen in between every time step. Thus, the training steps can not be computed on a GPU. Due to the frequent interruptions that are introduced by calls to RTLola, the speed of the computation is not just reduced by the time RTLola claims, but through the disability of using GPUs.

On the other hand, RTLola introduces expert knowledge to the neural network and simplifies the overall training problem. Thus, the prediction quality can be improved. The trade-off depends on the learning problem and the significance of additional knowledge introduced. The trade-off for our specific use case is observed in the next chapter.

RTLola Specifications

We use two kinds of specifications for the evaluation. One computes the displacement vector of the 3-dimensional position and the 4-dimensional orientation data. This specification is referred to as *displacement* and is given in Algorithm A.1.

The second specification computes the velocity as an integral over the acceleration sensor data. The integral is numerically approximated by an trapezoidal integral

$$\sum_i \frac{f(x_i) + f(x_{i-1})}{2} * (x_i - x_{i-1}),$$

where x_i and $f(x_i)$ are the i-th time step and senor value respectively. We refer to this specification as *integral*. The specification is given in Algorithm A.2.

Chapter 4

Evaluation

In this chapter we compare the different implementations. The introduced neural network models performances and the training hyperparameters are analysed. Due to the observation that local predictions produce very natural looking trajectories, we introduce an additional model to combine displacement vector predictions and absolute predictions. The new model is evaluated together with the other architectures.

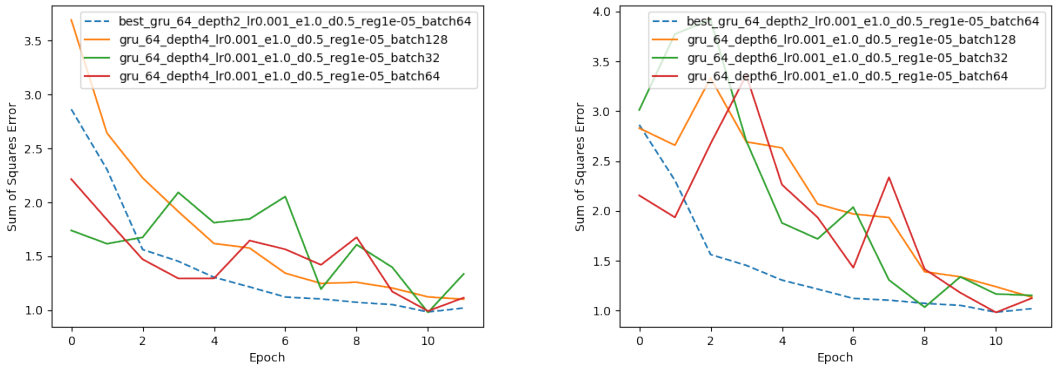
4.1. Implementation

In this section we introduce the practical details and used tools. The main parts are implemented in *Python3.6*, including the data generation and neural network wrappers for training/prediction.

For data generation, *AirSim*'s python interface is used to control and sample the flight simulations. The sampled traced are then saved in the *json* data format. The overall training set consist of 15000 piecewise and 20000 interpolated random flights. Each trajectory corresponds to a 30second flight sampled in real-time. Thus, the overall training data consists of roughly 12 days worth of random flights and 17 million samples.

For efficient computations and memory representation of arrays and tensors, we used the *Numpy* python module. This includes the preprocessing and data handling in general as training data is loaded directly into Numpy arrays. *Keras* and *TensorFlow2.0* are used for machine learning related tasks. TensorFlow offers automated gradient computations and out-of-the box neural network implementations for recurrent neural networks, such as GRU and LSTM. We also make use of TensorFlow's computation optimizations. Thus, training functions are compiled into optimized computation graphs. For example the scheduled sampling implementation used to train the recurrent neural networks is implemented this way. The code is appended in Algorithm A.3.

The *RTLola* monitoring tool is implemented in Rust. To use RTLola from our python code base, we implement a foreign function interface(*ffi*) to bridge the two languages. We implement this connection using the *cpython* module and a Rust interface for C



(a) Comparison with GRUs of depth 4

(b) Comparison with GRUs of depth 6

Figure 4.1.: A comparison of the number of GRU layers and different batch sizes. The plot depicts the valuation error during training of the models. In (a) and (b) we see a comparison with networks of depth 4 and 6, respectively. The dashed line corresponds to the best network *best-gru* with a depth of 2 and a batch size of 64.

function calls. Thus, the low-level communication is based on *C function calls* and uses *C structs* to pass data.

The networks were trained on two machines, i.e., a server with 8 CPU-cores and 32GB of RAM, and a laptop with 4 CPU-cores and 8GB of RAM.

4.2. Models

As introduced in Chapter 3, we use three different architectures, namely buffered feed-forward, recurrent and RTLola integrated neural networks. We separate the recurrent neural networks further into GRU and LSTM architectures. The RTLola integrated networks are also based on recurrent neural networks and use the same architectures. Due to the slower training time of RTLola integrated architectures, we rely on the basic recurrent architectures for the hyperparameter search. The hyperparameters that yield promising results on the basic recurrent neural networks are used for the RTLola integrated versions.

All architectures share the same learning rate $\alpha = 0.001$, number of hidden units $h = 64$, and regularization $\lambda = 10^{-5}$. The recurrent neural networks additionally require the scheduled sampling parameter ϵ and dropout parameter $p_d = 0.5$. Due to the GRU networks having less parameters, they can be trained faster in comparison to LSTMs. Thus, we use the GRU architecture as guiding model for the hyperparameter search.

Depth and Batch Size

In Figure 4.1, we compare GRU neural networks with different hyperparameters. The figure depicts the valuation error after each training epoch. A faster and continuously

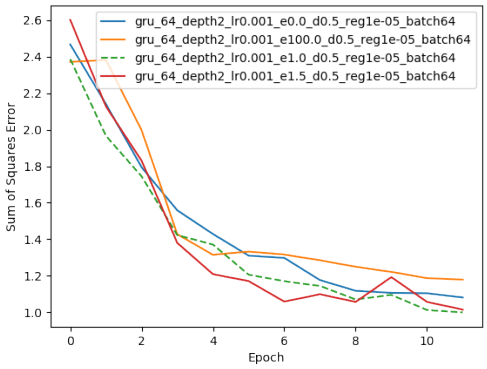


Figure 4.2.: A comparison of different scheduled training parameters. The best results are achieved with $\epsilon \in \{1.0, 1.5\}$. However, the loss with $\epsilon = 1.0$ is more consistently decreasing.

decreasing valuation loss corresponds to a good model. We pick the best version, depicted by a dashed line, and the corresponding hyperparameters. The model *best-gru* corresponds to a network of depth 2, i.e., two layers with 64 GRU units. The network was trained with a batch size of 64.

Scheduled Sampling

To determine an appropriate scheduled sampling parameter ϵ, we compare different values on the GRU architecture. In Figure 4.2, the valuation error of multiple values are compared. During training, the values are linearly interpolated between 0.25 and the given ϵ. Formally,

$$\hat{\epsilon}_i = 0.25 + \frac{i}{e} * (\epsilon - 0.25)$$

$$\epsilon_i = \min\Big(1.0, \max(0.0, \hat{\epsilon}_i)\Big),$$

where i and e are the current and total epoch, respectively. Note that the model with $\epsilon = 0.0$ simply doesn't use scheduled-sampling at all and $\epsilon = 100$ refers to always reusing the predictions. We can see that the best results are archived with $\epsilon \in \{1.0, 1.5\}$. Thus, we use both values during training. However, we observe that $\epsilon = 1.5$ (or even higher) produces better results with increasing number of training iterations. For example, the best model was trained over 200 epochs and $\epsilon = 4.0$.

Combined models

During the evaluation, we observe that the prediction of displacement vectors without any knowledge about the absolute position produces the best results. We introduce a new model that combines the network's displacement vector predictions and the absolute position by an explicit sum as depicted in Figure 4.3. This way the network predicts

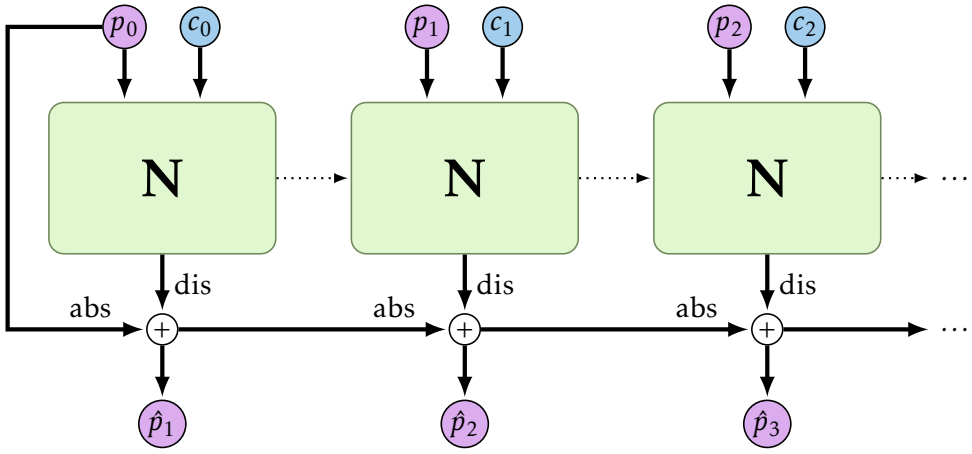


Figure 4.3.: A combination of displacement vector and absolute position prediction. The network predicts displacement vectors that are explicitly summed up to absolute positions.

displacement vectors while the explicit sum introduces knowledge about the absolute position. Note that during training of this new model, the scheduled sampling method not only randomly chooses the inputs to the network but also the input to the explicit sum. When the training data is used, both the input to the network and the absolute position input to the sum is the position from the training data. During inference, the predicted value is reused as input to the network and the explicit sum. Given our implementation of scheduled sampling where gradients are forwarded beyond the random choice, the new sums additionally introduce connections for the gradient to flow directly to earlier time steps. The new architecture is referred to as *combination* in the result tables.

4.3. Results

In Table 4.4, the evaluation results for piecewise trajectories are depicted. For absolute position predictions, RTLola effectively reduces the loss. In comparison to the other models within this class of predictions, RTLola integration even achieves the lowest loss. Due to the use of RTLola and the corresponding Python-Rust interface, the optimized computational graphs of TensorFlow could not be applied. Thus, the training process is slower and requires up to 270 minutes.

When comparing the absolute position predictions with the displacement vector predictions, even the RTLola variants are outperformed. The prediction of displacement vectors without any global information outperforms the other approaches and produces very realistic looking trajectories. Overall, these predictions achieve the lowest loss of 30540. Note that the loss is computed on the predicted trajectory after the corresponding absolute positions are computed, and thus, is a fair comparison.

The introduced combined architecture with RTLola further improves the prediction quality of RTLola integrated architectures. The corresponding loss almost matches

		time		
model	spec	train[min]	predict[s]	loss
Absolute Position				
BFF		≤1	≤1	**83141**
GRU		23	76	**101547**
LSTM		23	75	101981
GRU + RTLola	displacement	272	246	**74061**
GRU + RTLola	integral	270	246	92299
LSTM + RTLola	displacement	88	191	88792
LSTM + RTLola	integral	271	244	78044
Displacement Vector				
BFF		≤1	≤1	41173
GRU		24	148	**30540**
LSTM		24	135	32073
Combination				
LSTM + RTLola	displacement	343	120	35568
LSTM + RTLola	displacement	1370	126	**32440**
LSTM + RTLola	integral	145	115	89226
GRU + RTLola	integral	176	118	99118

Figure 4.4.: The test errors of trajectory predictions from piecewise controls. All networks were evaluated on the same test set. The displacement vectors predictions are summed up to compute the corresponding absolute positions. This happens only for the prediction and not during training, thus, the model has no information about absolute positions. In contrast to that, the combination architecture has an explicit sum as part of the model, thus training with respect to the absolute position loss. The loss is the mean summed squared error over all trajectories of the test set.

		time		
model	spec	train[min]	predict[s]	loss
Absolute Position				
BFF		2	4	754566
GRU		352	694	**712010**
LSTM		173	905	723162
GRU + RTLola	integral	1358	1763	850211
LSTM + RTLola	displacement	1043	671	**703167**
Displacement Vector				
BFF		6	4	958658
GRU		236	707	472202
LSTM		175	708	**384863**
Combination				
GRU + RTLola	integral	802	645	**643238**

Figure 4.5.: The test errors of trajectory predictions from interpolated controls. The best predictions come from local displacement vector predictions without knowledge of the absolute positions. Absolute position prediction and combined predictions are slightly improved with RTLola. The interpolated traces consist of more samples, and thus, the training time and error is higher.

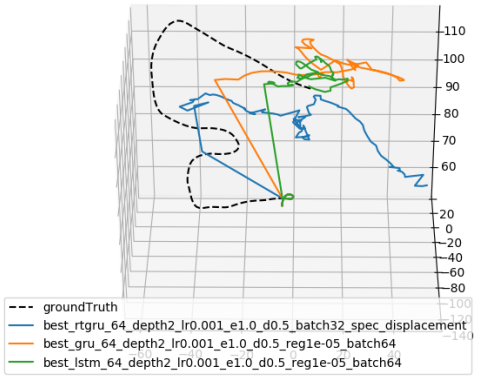
the overall best prediction with a loss of 32440. Comparing the different RTLola specifications, the best results are achieved with the *displacement* specification. Note that the second *LSTM + RTLola* was trained over more epochs, and thus, has a much longer training time. However, the loss only improved slightly despite the difference in training time.

The results for interpolated trajectories are depicted in Table 4.5. As mentioned in Section 3.1.2, the trajectories that correspond to interpolated control signals consist of 512 samples. This are exactly four times as many steps as the piecewise trajectories. Due to the accumulation error summing up over all time steps, the training problem is much harder. Furthermore, the loss function sums up the prediction errors along the trajectory, and thus, the loss is much higher compared to the piecewise trajectories.

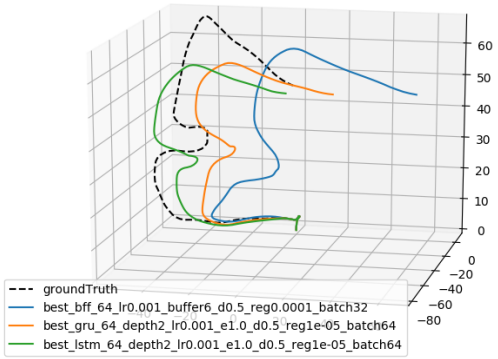
The evaluation of the different models shows similar results as in the piecewise trajectory prediction setting. Again, the overall best predictions are displacement vector predictions without any knowledge of the absolute positions. The combined architec-

ture with RTLola slightly improves the prediction quality of GRU/LSTM architecture. However, it was not possible to match the displacement vector predictions as closely as for piecewise trajectories.

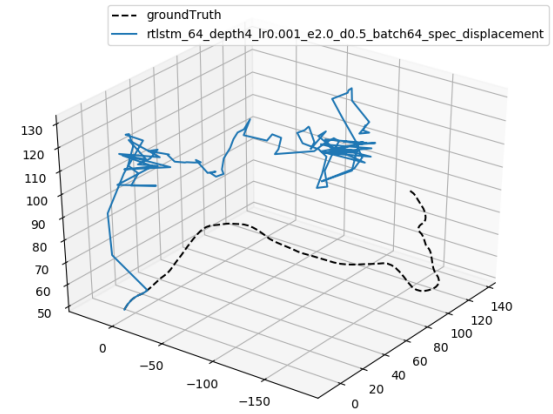
Prediction Quality In Figure 4.6, the predictions of all architectures are compared aside the computed test loss. The absolute position predictions in Figure 4.6a approximate the overall flow of the trajectory. However, the predictions are very fuzzy due to the loss function only accounting for the absolute difference and not local changes in behavior. In comparison to the displacement vector predictions in Figure 4.6b where only local predictions are used, absolute predictions look very unnatural. Predicting the local changes and penalizing local divergence instead of the absolute position produces very smooth and natural looking trajectories. Thus, displacement vector predictions not only have a lower prediction loss but also produce more natural trajectories. The combination architectures as depicted in Figure 4.6c and Figure 4.6d achieve a loss similar to the displacement vector predictions. However, when comparing the results, the predicted trajectories resemble the unnatural appearance of absolute position predictions. The trajectories remain close to the original trace but look very fuzzy with many abrupt changes in direction.



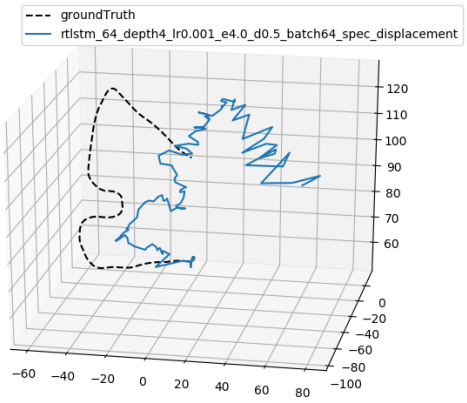
(a) Absolute Position



(b) Displacement Vector



(c) Combined Architecture



(d) Combined Architecture - Longer Training

Figure 4.6.: Trajectory predictions of the different architectures. The dotted line in each picture represents different ground-truth trajectories. The corresponding control signals are passed to the networks and the trajectories are predicted. The best predictions come from displacement vector predictions (b). The combined predictions (c, d) improve on the pure absolute position predictions (a). However, the produced trajectories are still very fuzzy and do not look natural.

Chapter 5

Conclusion & Further Work

Multi-step trajectory prediction is a hard learning problem. In this thesis, we trained and compared different neural network architectures. We proposed the introduction of expert knowledge by the means of RTLola specifications to improve the prediction quality. The evaluation shows that the RTLola integration helps the network and the test loss could be decreased. Additionally, we observed that local predictions, i.e., displacement vectors, produced very natural looking trajectories. This insight was used to construct and evaluate a new architecture to combine the local predictions with an explicit sum on the lowest layer. The sum is part of the network and adds up the local predictions over time to compute the absolute position. The connections introduced by the sum are direct connections between time steps. Due to an adapted scheduled sampling implementation, the gradients are also forwarded along the new connections. Thus, gradients computed on the lowest layer can be directly forwarded to adjacent time steps without first passing through the network layers.

For further work, we propose the use of more involved RTLola specifications as well as trajectory prediction for systems with already existing RTLola specifications. The existing specifications might be reused for the prediction task without any additional efforts of coming up with useful specifications. Additionally, the training of recurrent neural networks is hard and there are many hyperparameters that can be adjusted to improve the training process and prediction quality. The results might be further improved by comparing models on different sets of hyperparameters and training the networks over longer periods of time. Furthermore, the absolute position predictions look very unnatural and fuzzy. One could penalize the abrupt changes explicitly by adapting the loss function. The additional term in the loss function acts like regularization and the predicted traces could be smoothened.

Bibliography

[1] K. S. Narendra and K. Parthasarathy. 1990. Identification and control of dynamical systems using neural networks. *IEEE Transactions on Neural Networks*, 1, 1, 4–27. ISSN: 1941-0093. DOI: 10.1109/72.80202.

[2] C.A.L. Bailer-Jones, D.J.C. MacKay, and Philip Withers. 1998. A recurrent neural network for modelling dynamical systems. *Network (Bristol, England)*, 9, (December 1998), 531–47. DOI: 10.1088/0954-898X/9/4/008.

[3] A. Punjani and P. Abbeel. 2015. Deep learning helicopter dynamics models. In *2015 IEEE International Conference on Robotics and Automation (ICRA)*, 3223–3230. DOI: 10.1109/ICRA.2015.7139643.

[4] Alexander Rehmer and Andreas Kroll. 2019. On using gated recurrent units for nonlinear system identification. In (June 2019), 2504–2509. DOI: 10.23919/ECC.2019.8795631.

[5] Nima Mohajerin, Melissa Mozifian, and Steven Waslander. 2018. Deep learning a quadrotor dynamic model for multi-step prediction. In (May 2018), 2454–2459. DOI: 10.1109/ICRA.2018.8460840.

[6] Peter Faymonville, Bernd Finkbeiner, Malte Schledjewski, Maximilian Schwenger, Marvin Stenger, Leander Tentrup, and Hazem Torfah. 2019. Streamlab: stream-based monitoring of cyber-physical systems. In *Computer Aided Verification*. Isil Dillig and Serdar Tasiran, editors. Springer International Publishing, Cham, 421–431. ISBN: 978-3-030-25540-4.

[7] Alexander Parlos, Omar Rais, and Amir Atiya. 2000. Multi-step-ahead prediction using dynamic recurrent neural networks. In volume 13. (January 2000), 349–352. ISBN: 0-7803-5529-6. DOI: 10.1109/IJCNN.1999.831517.

[8] R Boné and M Crucianu. 2009. Multi-step-ahead prediction with neural networks: a review. 2, (January 2009).

[9] Nima Mohajerin and Steven Waslander. 2019. Multistep prediction of dynamic systems with recurrent neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, PP, (January 2019), 1–14. DOI: 10.1109/TNNLS.2019.2891257.

[10] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep learning*. MIT press.

[11] Tom Schaul, Ioannis Antonoglou, and David Silver. 2014. Unit tests for stochastic optimization. In *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*. Yoshua Bengio and Yann LeCun, editors.

[12] Diederik P Kingma and Jimmy Ba. 2014. Adam: a method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

[13] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. 2013. On the difficulty of training recurrent neural networks. In *International conference on machine learning*, 1310–1318.

[14] Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. 2009. Curriculum learning. In *Proceedings of the 26th Annual International Conference on Machine Learning* (ICML '09). Association for Computing Machinery, Montreal, Quebec, Canada, 41–48. ISBN: 9781605585161. DOI: 10.1145/1553374.1553380.

[15] Eugene L. Allgower and Kurt Georg. 1990. *Numerical Continuation Methods: An Introduction*. Springer-Verlag, Berlin, Heidelberg. ISBN: 0387127607.

[16] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural Comput.*, 9, 8, (November 1997), 1735–1780. ISSN: 0899-7667. DOI: 10.1162/neco.1997.9.8.1735.

[17] Kyunghyun Cho, Bart van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning phrase representations using RNN encoder–decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, Doha, Qatar, (October 2014), 1724–1734. DOI: 10.3115/v1/D14-1179.

[18] K. Greff, R. K. Srivastava, J. Koutník, B. R. Steunebrink, and J. Schmidhuber. 2017. Lstm: a search space odyssey. *IEEE Transactions on Neural Networks and Learning Systems*, 28, 10, 2222–2232. ISSN: 2162-2388. DOI: 10.1109/TNNLS.2016.2582924.

[19] 1988. *Learning representations by back-propagating errors*. *Neurocomputing: Foundations of Research*. MIT Press, Cambridge, MA, USA, 696–699. ISBN: 0262010976.

[20] P. J. Werbos. 1990. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78, 10, 1550–1560. ISSN: 1558-2256.

[21] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15, 1929–1958.

[22] Jan Baumeister, Bernd Finkbeiner, Maximilian Schwenger, and Hazem Torfah. 2019. Fpga stream-monitoring of real-time properties. *ACM Trans. Embed. Comput. Syst.*, 18, 5s, (October 2019). ISSN: 1539-9087. DOI: 10.1145/3358220.

[23] Shital Shah, Debadeepta Dey, Chris Lovett, and Ashish Kapoor. 2017. Airsim: high-fidelity visual and physical simulation for autonomous vehicles. *ArXiv*, abs/1705.05065.

[24] Epic Games. 2019. Unreal engine. Version 4.18. (2019). https://www.unrealengine.com.

[25] Gabriel Hoffmann, Haomiao Huang, Steven Waslander, and Claire Tomlin. 2007. Quadrotor helicopter flight dynamics and control: theory and experiment. In *AIAA guidance, navigation and control conference and exhibit*, 6461.

[26] Jiawei Han, Micheline Kamber, and Jian Pei. 2011. *Data Mining: Concepts and Techniques*. (3rd edition). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. ISBN: 0123814790.

[27] S. CHEN, S. A. BILLINGS, and P. M. GRANT. 1990. Non-linear system identification using neural networks. *International Journal of Control*, 51, 6, 1191–1214. DOI: 10.1080/00207179008934126.

[28] Tsungnan Lin, William Horne, Peter Tino, and C. Giles. 1996. Learning long-term dependencies in narx recurrent neural networks. *Neural Networks, IEEE Transactions on*, 7, (December 1996), 1329 –1338. DOI: 10.1109/72.548162.

[29] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Lstm can solve hard long time lag problems. In *Advances in neural information processing systems*, 473–479.

[30] Samy Bengio, Oriol Vinyals, Navdeep Jaitly, and Noam Shazeer. 2015. Scheduled sampling for sequence prediction with recurrent neural networks. In *Advances in Neural Information Processing Systems 28*. C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors. Curran Associates, Inc., 1171–1179.

Appendix A

Code

```
input pitch: Float64
input roll: Float64
input throttle: Float64
input yaw_rate: Float64
input pos_x: Float64
input pos_y: Float64
input pos_z: Float64
input ori_w: Float64
input ori_x: Float64
input ori_y: Float64
input ori_z: Float64
input acc_x: Float64
input acc_y: Float64
input acc_z: Float64
output dis_x:= pos_x - pos_x.offset(by:-1).defaults(to:0.0)
output dis_y:= pos_y - pos_y.offset(by:-1).defaults(to:0.0)
output dis_z:= pos_z - pos_z.offset(by:-1).defaults(to:0.0)
output dis_ori_w:= ori_w - ori_w.offset(by:-1).defaults(to:0.0)
output dis_ori_x:= ori_x - ori_x.offset(by:-1).defaults(to:0.0)
output dis_ori_y:= ori_y - ori_y.offset(by:-1).defaults(to:0.0)
output dis_ori_z:= ori_z - ori_z.offset(by:-1).defaults(to:0.0)
```

Figure A.1.: RTLola specification for the training. Accumulate velocity and compute displacements for position and orientation

```
input pitch: Float64
input roll: Float64
input throttle: Float64
input yaw_rate: Float64
input pos_x: Float64
input pos_y: Float64
input pos_z: Float64
input ori_w: Float64
input ori_x: Float64
input ori_y: Float64
input ori_z: Float64
input acc_x: Float64
input acc_y: Float64
input acc_z: Float64
output vel_x:= vel_x.offset(by:-1).defaults(to:0.0)
    + (acc_x + acc_x.offset(by:-1).defaults(to:0.0))/2.0 * time
output vel_y:= vel_y.offset(by:-1).defaults(to:0.0)
    + (acc_y + acc_y.offset(by:-1).defaults(to:0.0))/2.0 * time
output vel_z:= vel_z.offset(by:-1).defaults(to:0.0)
    + (acc_z + acc_z.offset(by:-1).defaults(to:0.0))/2.0 * time
```

Figure A.2.: RTLola specification for the training. Calculates the velocity as an numerical integral over acceleration sensors. The variable time is the time-interval between two predictions, and thus, depends on the sampling frequency.

```
LOSS = nets.util.SumOfSquaresLoss()

@tf.function
def train_step(model, inputs, target, epsilon, batch_size):
    """ implements accumulated learning training step """
    with tf.GradientTape() as tape:
        prediction_len = target.shape[1]
        prediction_dim = target.shape[2]
        predictions = tf.TensorArray(
            nets.data.TF_FLOAT,
            size=prediction_len,
            clear_after_read=False)

        # first prediction is explicit as it cannot depend on earlier predictions
        prediction = model(inputs[:, 0:1], training=True)
        predictions = predictions.write(0, prediction)

        # predict step-wise and accumulate results
        for i in tf.range(1, prediction_len):
            # roll a dice
            decision = tf.logical_and(
                tf.less(tf.random.uniform([1]), epsilon), # with prob<epsilon, reuse old
                    prediction
                tf.greater(i, 16)) # except for the first steps (model is still adjusting)
            inp = tf.where(
                decision,
                tf.concat(
                    [inputs[:, i:i+1, :-prediction_dim], predictions.read(i-1)],
                    axis=2),
                inputs[:, i:i+1])
            prediction = model(inp, training=True)
            predictions = predictions.write(i, prediction)

        # transform tensorarray to tensor:
        # stacking yields shape=(prediction_len, batch_size, 1, dim_features)
        # -> transpose to restore correct shape=(batch_size, prediction_len, dim_features)
        prediction = tf.squeeze(tf.transpose(predictions.stack(), [1, 0, 2, 3]))

        loss = LOSS(target, prediction)
        grad = tape.gradient(loss, model.trainable_variables)
        # update model
        model.optimizer.apply_gradients(zip(grad, model.trainable_variables))
    return loss
```

Figure A.3.: Scheduled Sampling