ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

RSL Robotic Systems Lab
Prof. Dr. Marco Hutter

Semester Thesis

Learning Forward Models Of ANYmal Dynamics

Using Temporal Convolutional Networks

Spring Term 2019

Supervised by:
Vassilios Tsounis
Jonhoo Lee
Pascal Egli

Author:
Nikita Rudin

Declaration of Originality

I hereby declare that the written work I have submitted entitled

Your Project Title

is original work which I alone have authored and which is written in my own words.[1]

Author(s)

First name	Last name

Student supervisor(s)

First name	Last name

Supervising lecturer

Marco	Hutter

With the signature I declare that I have been informed regarding normal academic citation rules and that I have read and understood the information on 'Citation etiquette' (`https://www.ethz.ch/content/dam/ethz/main/education/rechtliches-abschluesse/leistungskontrollen/plagiarism-citationetiquette.pdf`). The citation conventions usual to the discipline in question here have been respected.

The above written work may be tested electronically for plagiarism.

Place and date	Signature

[1] Co-authored work: The signatures of all authors are required. Each signature attests to the originality of the entire piece of written work in its final form.

Intellectual Property Agreement

The student acted under the supervision of Prof. Hutter and contributed to research of his group. Research results of students outside the scope of an employment contract with ETH Zurich belong to the students themselves. The results of the student within the present thesis shall be exploited by ETH Zurich, possibly together with results of other contributors in the same field. To facilitate and to enable a common exploitation of all combined research results, the student hereby assigns his rights to the research results to ETH Zurich. In exchange, the student shall be treated like an employee of ETH Zurich with respect to any income generated due to the research results.
This agreement regulates the rights to the created research results.

1. Intellectual Property Rights

1. The student assigns his/her rights to the research results, including inventions and works protected by copyright, but not including his moral rights ("Urheberpersönlichkeitsrechte"), to ETH Zurich. Herewith, he cedes, in particular, all rights for commercial exploitations of research results to ETH Zurich. He is doing this voluntarily and with full awareness, in order to facilitate the commercial exploitation of the created Research Results. The student's moral rights ("Urheberpersönlichkeitsrechte") shall not be affected by this assignment.
2. In exchange, the student will be compensated by ETH Zurich in the case of income through the commercial exploitation of research results. Compensation will be made as if the student was an employee of ETH Zurich and according to the guidelines "Richtlinien für die wirtschaftliche Verwertung von Forschungsergebnissen der ETH Zürich".
3. The student agrees to keep all research results confidential. This obligation to confidentiality shall persist until he or she is informed by ETH Zurich that the intellectual property rights to the research results have been protected through patent applications or other adequate measures or that no protection is sought, but not longer than 12 months after the collaborator has signed this agreement.
4. If a patent application is filed for an invention based on the research results, the student will duly provide all necessary signatures. He/she also agrees to be available whenever his aid is necessary in the course of the patent application process, e.g. to respond to questions of patent examiners or the like.

2. Settlement of Disagreements

Should disagreements arise out between the parties, the parties will make an effort to settle them between them in good faith. In case of failure of these agreements, Swiss Law shall be applied and the Courts of Zurich shall have exclusive jurisdiction.

Place and date	Signature

Contents

Abstract **v**

Symbols **1**

1 Introduction **2**
1.1 Two Approaches of Reinforcement Learning 2
1.2 Related Work 3
1.2.1 Model-Based Reinforcement Learning 3
1.2.2 Temporal Convolutional Networks 4

2 Problem set-up **5**
2.1 Definition of Dynamics 5
2.2 Datasets 5
2.2.1 Simulator 5
2.2.2 Real Robot 6

3 One-Step Training **7**
3.1 Model Description 7
3.1.1 TCN 7
3.1.2 Stochastic Model 7
3.2 Inputs and Targets Processing 9
3.3 One-Step Predictions and Roll-Out 10
3.4 C++ Implementation for On-Policy Roll-Out 11
3.5 Results and Findings 12
3.5.1 One-Step Prediction 12
3.5.2 Roll-Out Performance 13
3.5.3 Importance of Noise 13
3.5.4 Mean Squared Error versus Log-Likelihood 16
3.5.5 Lack of Generalization Between Policies 17
3.5.6 Training on Combined Datasets 17

4 Masked Sequence Training **19**
4.1 Motivation 19
4.2 Description 19
4.2.1 On Policy Predictions 20
4.3 Mixture Density Network 21
4.4 Results 21
4.4.1 Mean Squared Error versus Log-Likelihood 23
4.4.2 Conditioning on the Initial Observation 24
4.4.3 Training on Combined Datasets 25

5 Discussion **26**
5.1 Two Training Approaches 26
5.2 Mixture Density Network Versus Single Mode 26
5.3 Stochastic Versus Non-Stochastic Model 26
5.4 Training On Richer Datasets 27
5.5 Potential future work 27

6 Conclusion **28**

Bibliography **29**

A **30**
A.1 One-step Training 31
A.2 Masked Sequence Training 33

Abstract

The focus of this project is to investigate the ability of neural networks to predict the dynamics of a highly complex system, such as the legged robot ANYmal. The main complexity of such systems comes from non-linearities introduced by contact forces. This is typically solved by computationally heavy physics simulators, which have the disadvantage of requiring a large number of fixed parameters and lack some important aspects such as actuator dynamics. The motivation behind this project is to substitute the simulator with a trained neural network, which would then be used in a model-based reinforcement learning set-up, where the learned model of the environment is used to train or improve a policy. The intermediate step of learning a model has the potential advantage of being much more sample efficient than learning a policy directly. It has been demonstrated that neural networks have the ability to model the environment of toy examples [1] and ATARI games [2], but there has been no application to complex problems such as a real world legged robot like ANYmal. We will show promising results, but also potential pitfalls of these models.

Symbols

Symbols

q	joint angles
$\dot{q}$	joint velocities
${}_B g$	measured gravity vector in the base frame
${}_W r_{WB,z}$	base height above the ground
${}_B v_{WB}$	base linear velocity
${}_B \omega_{WB}$	base angular velocity
τ_j^*	desired joint torques
s_k	state at time step k
o_k	observation at time step k
$\hat{o}_k$	predicted observation at time step k
a_k	action at time step k
σ	standard deviation (std)
f_θ	function f parametrized by learned parameters θ

Indices

x	x axis
y	y axis
z	y axis

Acronyms and Abbreviations

RSL	Robotic Systems Lab, ETH
TCN	Temporal Convolutional Network
RNN	Recurrent Neural Network
MDN	Mixture Density Network
VAE	Variational Auto Encoder
MPC	Model Predictive Control

Chapter 1

Introduction

1.1 Two Approaches of Reinforcement Learning

Model-free deep reinforcement learning is proving to be a powerful method for solving a multitude of tasks ranging from playing games to controlling robotic systems. It was recently used to control the ANYmal robot [3] and outperformed manually designed controllers both in terms of walking speed and computational efficiency. One drawback of this method is the fact that it requires a lot (potentially millions) of trials to learn a policy. This makes it nearly impossible to learn directly on a real world system. Training is usually performed in a simulator with the hope that the learned policy will generalize to the real system. Simulators can only capture a simplified version of the real dynamics so there is no guaranty that the learned policy will perform well in the real world. One solution is to randomize the simulated dynamics and train a policy that is robust to these changes. Another interesting option is to include a learned model as a part of the simulator. In the case of the ANYmal work [3], a neural network is used to predict joint torques, needed by the physics engine, from actuation commands. This allows the simulator to encompass complex actuator dynamics including springs and electronics that would be too complicated to model analytically. A tempting next-step is to replace the complete simulator by a learned model. The model is usually not fixed, but learned along with the policy. This brings us into the model-based paradigm.
Model-based reinforcement learning focuses on the idea that for some tasks it can be easier to learn a model of the environment than a policy. This model can then be used in different ways to obtain a policy including Monte-Carlo search and optimization based methods such as MPC. In both cases the model has to be general enough to capture very diverse and potentially unseen behaviours. As shown throughout this work, this task would be too ambitious for our system.
Finally, recent work [1],[2] focuses on a hybrid set-up first proposed by Sutton nearly 30 years ago [4]. This set-up combines both model-based and model-free approaches. The general algorithm is summarized in the following steps (starting from some policy):

1. Interact with the environment according to the policy and collect trajectories.
2. Train the model to predict transitions and rewards in the collected data.
3. Improve the policy by having it interact with the model and go to step 1.

This set-up has some important advantages:

- It doesn't require a linearisation of the model, which is needed by most optimization based approaches.

- Once trained, the policy can be used without a model in a computationally efficient way.
- The model doesn't need to generalize to all sort of behaviours, but must only be accurate around the current iteration of the policy.

The relaxed generalization requirement is extremely important in our case. As shown later, it is extremely difficult to ensure that a neural network learns the dynamics in the physical sense. It is, however, often able to make good predictions for a particular policy. Potentially, such a model would be sufficient in a Dyna-type set-up.

1.2 Related Work

1.2.1 Model-Based Reinforcement Learning

Even though the Dyna set-up was proposed back in 1991, it has regained popularity in the past few years. Notably it is used in the following three cases:

World Models [1] This paper teaches an agent to play two simple (Openai gym 2D car racing and Vizdoom) games from video input. They use a variational auto-encoder (VAE) together with a recursive mixture density network (MDN-RNN) as a model. At each iteration, the VAE is trained, then the MDN-RNN is trained. Finally the agent represented by a single linear layer is trained using a variant of an evolution strategy, which is claimed to work well for small parameter spaces. The described algorithm achieves state of the art results in the car racing task. It also achieves to solve the VizDoom task even though the model fails to capture some key aspects of the environment. It is claimed that an agent which performs well in the noisy inaccurate model, will perform even better in the cleaner real environment. Interestingly, the paper motivates the approach by findings of neuroscience studies, which claim that a similar process happens in a human brain. Humans seem to learn a simplified model of the world and are able to predict how an action will affects our surroundings. We can then evaluate actions by comparing their predicted outcomes.

Model-Based Reinforcement Learning for Atari [2] This work follows closely the Dyna set-up and applies it to a variety of Atari games, with again videos as input. A complex stochastic model combining embeddings, discrete variables, encoders, attention and recurrent layers is used to predict the next frame of the game. The scheduled sampling technique [5], in which predicted samples progressively replace true data is used to improve the generalization power. This ensures that the model doesn't diverge once it is given its own predictions as input, even though the predictions might be different (i.e. blurry and noisy) from the true data. The algorithm achieves to match or even beat state of the art model-free approaches while using up to 22x less samples from the true environment. While the algorithm works well on most games, it completely fails on some particularly complex ones.

Wayve Autonomous Driving [6] The autonomous driving start-up Wayve has applied the world models algorithm to teach a car how to drive autonomously through empty countryside roads. A model predicts the next images captured by the front-facing camera. These prediction are used to train an agent. Once again the model combines a VAE and a probabilistic recurrent network. No numeric metrics are provided, but the car was successfully able to drive through different weather conditions. It is interesting to look at the reconstructed predictions of the

model. We can see that they are very blurry and far from fully representing the real world. Nevertheless, it is enough to train a policy that can be applied on the real car. This shows that in the Dyna/World models set-up we do not need an ideal representation of the environment.

1.2.2 Temporal Convolutional Networks

WaveNet: A Generative Model for Raw Audio [7] An architecture close to the TCN was introduced in this work from Google. A neural network relying on causal temporal convolutions is used to generate raw audio waveforms of human speech. It achieves to keep long-term realism and consistency using dilated convolutions. Combining the temporal convolutions with other more sophisticated techniques, the final architecture is highly complex and not easily reproducible. The result is the generation of realistic speech in multiple languages

An Empirical Evaluation of Generic Convolutional and Recurrent Networks for Sequence Modelling [8] The WaveNet architecture is adapted and highly simplified in this work. The final model called *Temporal Convolutional Network (TCN)* relies on blocks of dilated causal temporal convolutions. The main objective of this work is to present the TCN as a strong alternative to recurrent networks. The model is compared to various RNN architectures on different classical RNN benchmarks. The results suggest that the TCN outperforms most RNN architectures on a variety of tasks where temporal dependencies are important. The main advantages of the model are presented as follows:

- Parallelism of convolutions, which lead to much faster training, especially on a GPU.
- Controlled memory of the network through the receptive field
- Stable gradients, which allow for easier training
- Lower memory requirements during training because there is no need to store the intermediate hidden states of the RNN cells.

Chapter 2

Problem set-up

2.1 Definition of Dynamics

In usual reinforcement learning problems the environment dynamics are defined as function f where:

$$s_{k+1} = f(s_k, a_k)$$

In the case of a mobile robot, the true state is unknown to the robot and is actually undefined. We can only use observations, so the model has to approximate the following function:

$$o_{k+1} = f(o_k, a_k)$$

A single observation does not contain enough information to predict the next. For example, actuators have a delay that needs to be taken into account. Therefore the model receives a time window of previous observations and actions. The final function modeled by the network is the following:

$$o_{k+1} = f_\theta(o_{k-N}, \ldots, o_k, a_{k-N}, \ldots, a_k)$$

where θ are the parameters of the network and $N+1$ is the size of the time window. In this work observations are defined as:

$$o = [q,\ \dot{q},\ {}_B g,\ {}_W r_{WB,z},\ {}_B v_{WB},\ {}_B \omega_{WB}]^\top$$

and actions are:

$$a = \tau_j^*$$

2.2 Datasets

2.2.1 Simulator

The first approach used to collect data relies on a physics simulator combined with pre-trained RL policies. We used three policies:

- Walking policy
- Interesting policy combining walking and galloping (called *bigait* in this work)
- Random policy tuned to avoid falling

A dataset containing 150k transitions at 200Hz was recorded for each policy, along with a combined dataset. All datasets contain 300 trajectories of 500 transitions. Each trajectory starts from a randomized initial state and actions contain noise to increase diversity. Please note that the bigait policy can only handle little randomization without falling, as such the data collected using it is not as diversified as the rest.

2.2.2 Real Robot

In order to obtain richer data the second approach was to use the real ANYmal robot controlled by engineered controllers. These controllers allow it to walk at different speeds, in different directions and using different gaits. In particular the following gaits were used: stand, crawl, trot, flying trot and amble. One complication of the real world data is that the base height ${}_{W}r_{WB,z}$ is not a direct measurement, but is inferred from contacts and kinematics. It is not reliable and even after removing long term drift and bias, it drifts up during walking phases and comes back down while standing. An unreliable base height can be a severe problem when learning the dynamics because the contacts can not be predicted without knowing it precisely. In total 144k transitions were collected in 12 trajectories of 12k transitions each. The data was pre-processed to extract the observations described above and improve the base height measurement.

All datasets are saved as a csv file. All trajectories are concatenated vertically. Each row represents one set of observations and actions (o_k, a_k).

Chapter 3

One-Step Training

This chapter describes the main algorithmic approach used to train our model.

3.1 Model Description

3.1.1 TCN

Based on the hypothesis that information from multiple previous time steps is required to make a prediction into the future, we need to give our model a memory mechanism. The Temporal Convolutional Network was selected based on the advantages presented in the previous chapter. Compared to a simple dense network (MLP) the convolution mechanism allows us to have a large input window without increasing the number of parameters.
A TCN network is based on 1D convolutions where the convoluted axis is interpreted as time. These convolution layers are causal, which means that each node can only look back in time into the previous layer. This is implemented by simply padding with enough zeros on the left of each layer. Causality is available as a padding option in the Keras API, where it is described as: *output[t] does not depend on input[t+1:]*. Furthermore the convolutions can be dilated. Dilated convolutions have a stride between each input of the kernel. Effectively this multiplies the receptive field by the dilation factor. Finally the (dilated) causal convolutions are arranged in blocks. Each block contains two layers in parallel with a skip connection. In our case, the dilations are not used because no long term memory is required. Figure 3.1 is a graphical representation of the concepts of causality and dilation. It is important to note that the weights of the kernel are shared across a layer, so the number of parameters does not depend on the sequence length.

3.1.2 Stochastic Model

In this work we will investigate the advantages of a stochastic model over a fixed prediction. In particular the model predicts a state-dependant diagonal Gaussian distribution. At each time step, each component of the predicted observation has a mean $\hat{o}_{i,k}$ and standard deviation $\hat{\sigma}_{i,k}$. In terms of implementation, the output of the TCN is passed through two separated layers. One for the mean with no activation and one for the covariance with an exponential activation to ensure that $\hat{\sigma}_{i,k} > 0$. These two layers are convolutions with a kernel size of 1, which means that the same dense layer is applied to each time-step individually. For the non-stochastic model a simple mean-squared error loss is used. In the stochastic case

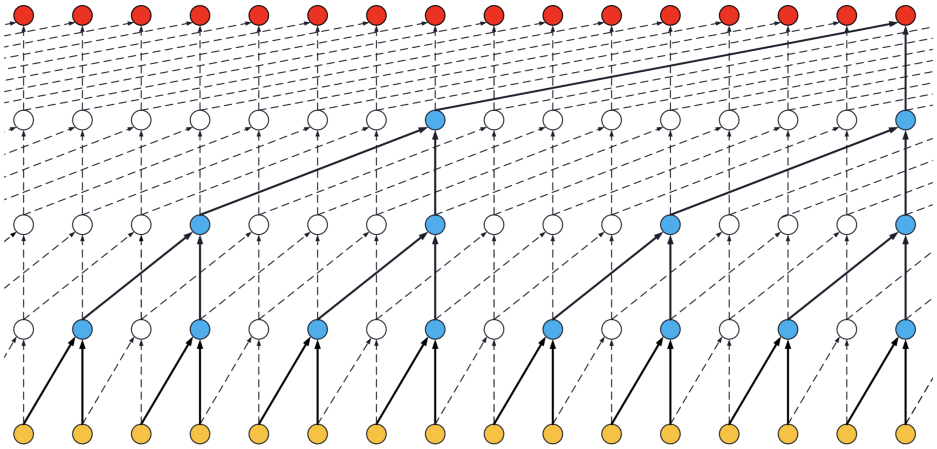


Figure 3.1: Graphical representation of the causality and dilation of the TCN layers. Here a kernel size of 2 and dilations of 1,2,4,8 are used.
Source: https://deepmind.com/blog/article/wavenet-generative-model-raw-audio

the negative log-likelihood loss can be derived as follows:

$$\mathcal{N}(y|\mu,\Sigma) = \frac{1}{(2\pi)^{D/2}|\Sigma|^{1/2}} \exp(-\frac{1}{2}(y-\mu)^\top \Sigma^{-1}(y-\mu))$$

$$-\ln(\mathcal{N}(y|\mu,\Sigma)) = \frac{D}{2}\ln(2\pi) + \frac{1}{2}\ln(|\Sigma|) + \frac{1}{2}(y-\mu)^\top \Sigma^{-1}(y-\mu)$$

for a diagonal covariance we have

$$\Sigma = \begin{pmatrix} \sigma_1^2 & & 0 \\ & \sigma_2^2 & \\ & & \ddots \\ 0 & & \sigma_D^2 \end{pmatrix}$$

$$\ln(|\Sigma|) = \sum_{i=1}^{D} \ln(\sigma_i^2)$$

$$(y_i-\mu_i)^\top \Sigma^{-1}(y_i-\mu_i) = \sum_{i=1}^{D}(y-i-\mu_i)^2/\sigma_i^2$$

finally:

$$-\ln(\mathcal{N}(y|\mu,\Sigma)) = \frac{D}{2}\ln(2\pi) + \sum_{i=1}^{D}\frac{1}{2}\ln(\sigma_i^2) + \frac{1}{2}\frac{(y_i-\mu_i)^2}{\sigma_i^2}$$

where D is the dimension of y.
The complete model is shown in figure 3.2. The TCN implementation is a slightly modified version of an open-source Keras code-base [9]. The *RELU* activations has been changed to *ELU* in order to allow easier learning of negative outputs. A L2 kernel regularizer has also been added. The number of layers, sequence length, dropout rate and regularizer coefficient are hyper-parameters that have been optimized. The final results do not use any dropout or L2-regularizer since the addition of noise proved to be more effective.

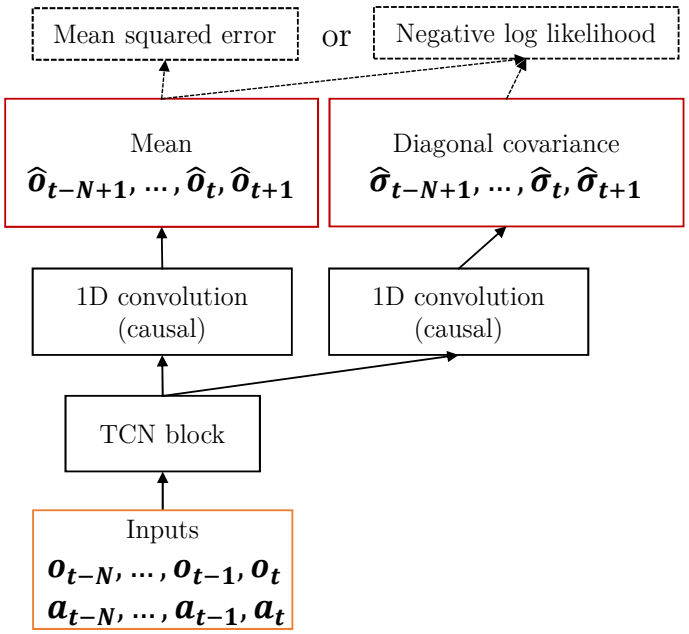


Figure 3.2: Complete model with fixed or stochastic output. The 1D convolution producing the mean has a linear activation, while the one producing the covariance has an exponential activation.

3.2 Inputs and Targets Processing

As described above, the model receives a time window of observations and actions. The TCN model allows to do sequence to sequence training, which means that the output is also a time window. In our case the target output is the input window shifted by one time steps into the future. Since the model needs a history of inputs for good predictions, for training can only outputs with enough memory can be used. Since the TCN has known receptive field, we can enforce this by setting the input window size as twice the receptive filed and only use the second half for training. The receptive field for B blocks with kernel size k and dilations d can be calculated as:

$$RF = \sum_{b=1}^{B} 2(k-1)d_b$$

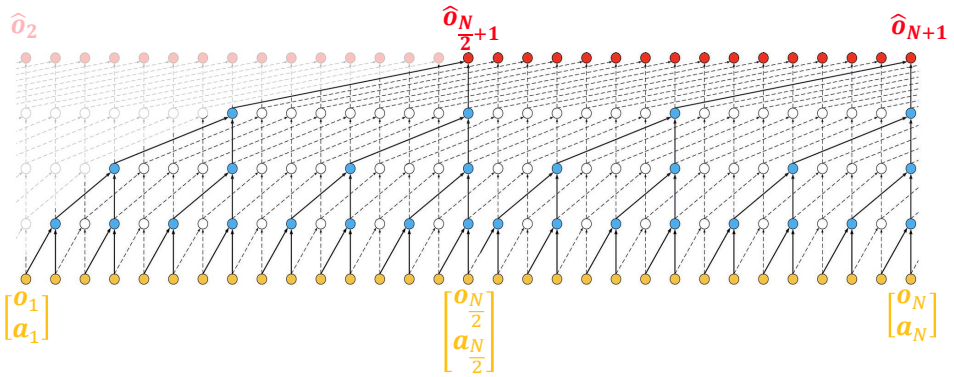


Figure 3.3: The output targets are a window of observations shifted by one with respect to the inputs. Because of causality, each output is a prediction one step into the future from the inputs it is dependent on. Only the second half of outputs $[\hat{o}_{N/2}, ..., \hat{o}_N]$ is used for training.
Modified from: https://deepmind.com/blog/article/wavenet-generative-model-raw-audio

3.3 One-Step Predictions and Roll-Out

It is important to distinguish between the way this model is trained and the way it would be used in a reinforcement learning set-up. During training the model always receives true observations and actions. Each output is a prediction one step into the future. Once trained, the model is used to make long predictions recursively. The last predicted observation is inserted into the input and used for the next prediction. We call this procedure a roll-out. Clearly, a good roll-out performance is more challenging than good one-step predictions because errors will accumulate and the predicted trajectory can diverge. This is a known problem in sequence-to-sequence learning with RNNs. Recurrent networks can potentially be trained directly on roll-outs, but this leads to training instabilities. A well-known technique called *Teacher forcing*[10] replaces the outputs of RNNs by the ground truth at each step. This is equivalent to a one-step prediction. It is also known that models trained in this way are fragile once used in open-loop (as is the case in roll-outs). *Scheduled sampling* [5] has been proposed as way of making the model more robust. Outputs of the model progressively replace ground truth inputs in order to force the model to handle its own mistakes correctly. This technique is used and described as important by Kaiser[2]. In this work it has been implemented, but has not shown any empirical benefits. It is unclear if this is due to a different problem set-up or simply an imperfect implementation.
Figures 3.4 and 3.5 show the two cases.

As a simplification, the actions given during the roll-out are the true actions that were used in the target trajectory. In an actual model-based reinforcement learning scenario, the next predicted observation $\hat{o}_{k+1}$ is given to the policy which in turn selects the corresponding actions a_{k+1}. This third scenario that we call on-policy roll-out is shown in figure 3.6

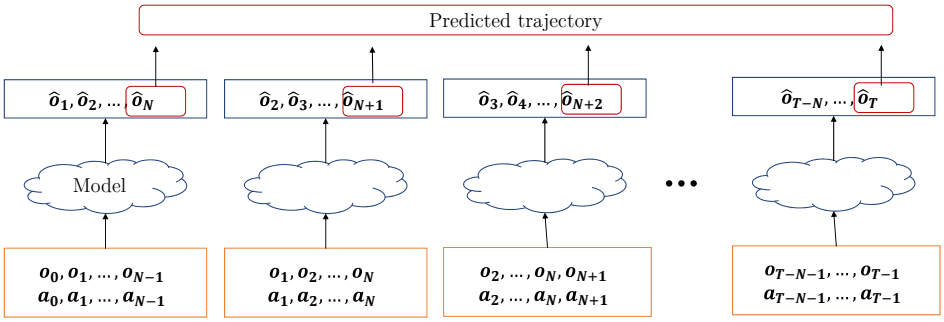


Figure 3.4: One-step prediction used for training

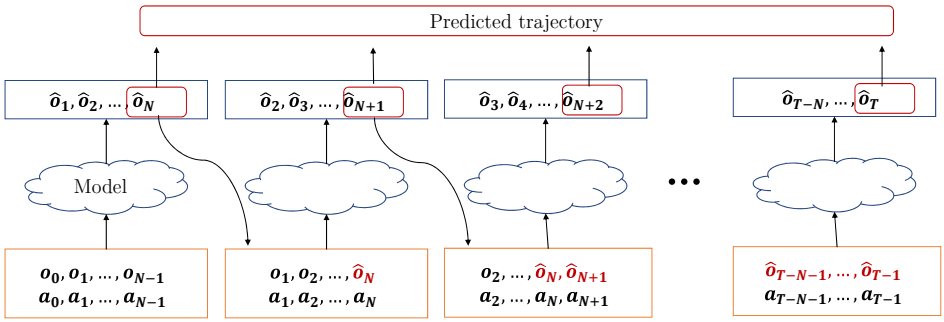


Figure 3.5: Roll-out used for testing

3.4 C++ Implementation for On-Policy Roll-Out

The simplification described above is necessary because the main implementation (in Python) does not allow to query the policy. For this reason and for potential deployment on the real robot, the model was implemented in RSL's in-house C++ package *Noesis*. This implementation allows to either train a model from scratch with any policy or use an existing dataset for pre-training. Pre-training can also be used to verify transfer learning capabilities between policies. The performance can be verified visually by playing the predictions in the visualizer. Unfortunately it lacks support for plot creation and as such, only qualitative results can be provided in this report. Videos are provided as supplementary material.
This implementation allows to visualize the training progress. In the current set-up, at each iteration, a new trajectory representing 5 seconds of data and 1000 samples, is collected and added to the estimator memory. The estimator is then re-fitted for 1 epoch. We can observe that around 40-50 iterations are needed in order to obtain good predictions. This represents 3-4 minutes of real time data. Hence, far less data is needed to train a model than what is needed to train a policy in a model-free set-up, where days of real-time data are used. A policy can then be learned using this model.

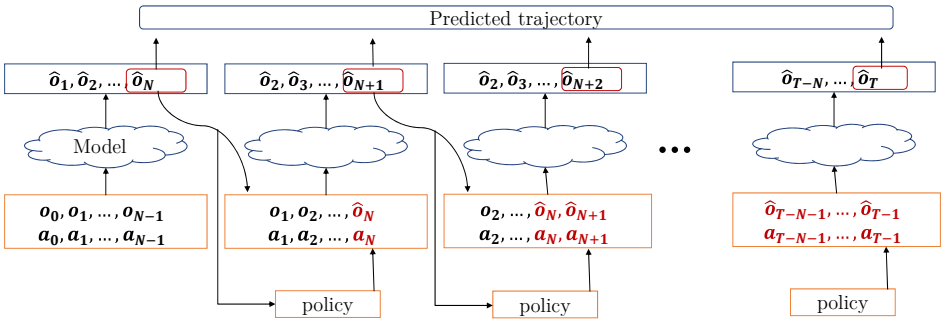


Figure 3.6: On-policy roll-out used in a model-based reinforcement learning algorithm.

Furthermore we can evaluate the computational efficiency of our model. Table 3.1 presents inference times for different batch sizes on a CPU and a GPU. We can see that with a powerful GPU and a large batch size, the model is very fast. Note that a batch size larger than 1 requires proper parallelization handling in the algorithm using this model.

	$[milliseconds/batch]$	$[samples/s]$
CPU - batch of 1	4,5	223
CPU - batch of 128	78.5	1625
GPU - batch of 1	2.8	358
GPU - batch of 128	3.4	37'380
GPU - batch of 512	6.5	78'745

Table 3.1: Mean inference time per batch as well as the number of samples treated per second ($\frac{batchsize}{time\ per\ batch}$). Clearly a larger batch size is advantageous. CPU:Intel Core i5-6300U @2.40GHz×4. GPU:RTX 2060 6GB.

3.5 Results and Findings

3.5.1 One-Step Prediction

As a first result we can look at the quality of one step predictions for the walking policy in the simulator. Plots in figure 3.7 present the predictive power on a selection of observations' components. Table 3.2 summarises the mean absolute errors along validation trajectories for walking and random policies in the simulator.
While these results can look impressive, it is important to remember that one time step is only 5ms and as such, $o_{k+1} \approx o_k$. Since the model has access to o_k it is not hard to have a good estimate of o_{k+1}.As a comparison predicting $\hat{o}_{k+1} = o_k$ without noise would lead to an average error of 0.2. Clearly, even if it can produce seemingly good results we do not want our model to learn a simple identity function. Therefore, it is important to look at the roll-out performance and not only the one-step predictions.

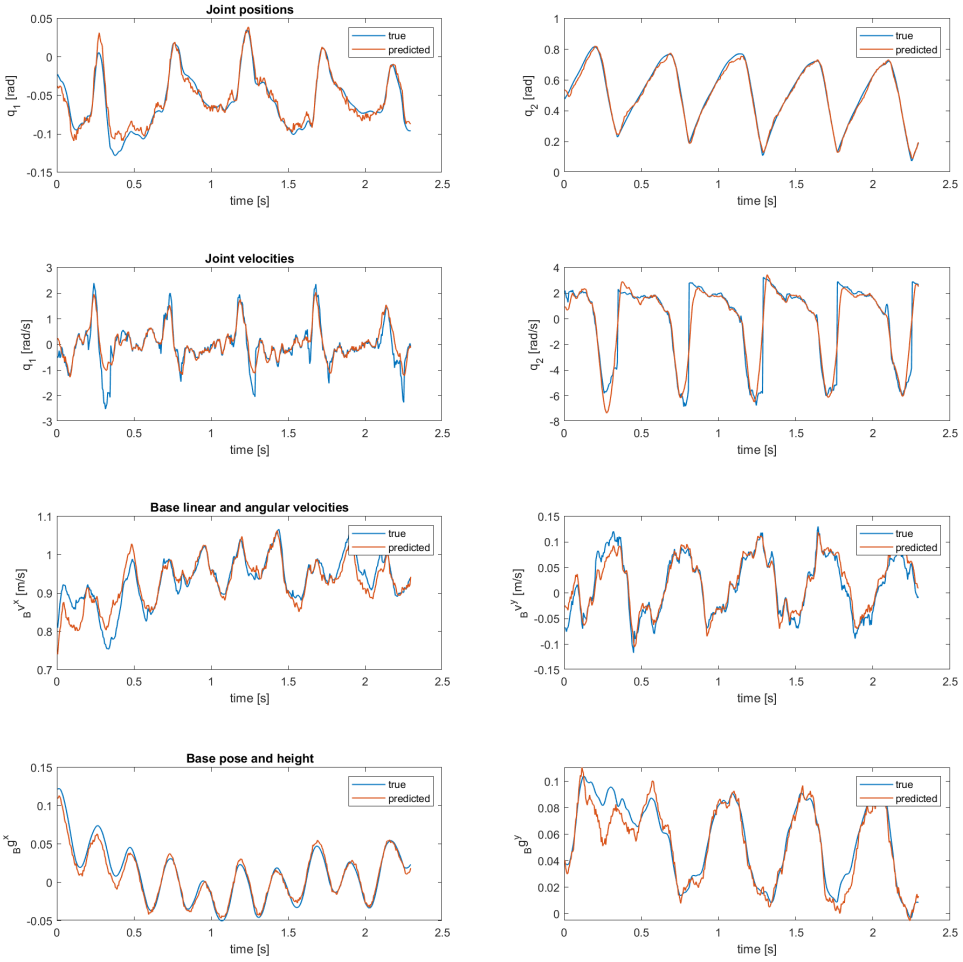


Figure 3.7: One-step predictions of a validation trajectory for selected components (4 joint positions, 2 joints velocities and x,y components of the gravity measurement are selected as a representative sample of the 34 dimensional observation). Stochastic model with 0.5 σ noise, walking policy.

3.5.2 Roll-Out Performance

Plots in figure 3.8 demonstrate the performance of the model in the roll-out scenario (with true actions). We can clearly see that the errors are larger that in the previous case. Nevertheless, the model is not simply outputting $\hat{o}_{k+1} = o_k$ and has learned at least some of the dynamics. Again, table 3.3 summarises the roll-out mean absolute errors along validation trajectories for the same policies.

3.5.3 Importance of Noise

One major finding of this approach is the importance of adding noise to the inputs. Noise is added dynamically whenever a new batch is generated. As expected noisy inputs lead to a higher one-step training loss, but interestingly, the roll-out error decreases dramatically. Training on noisy inputs stabilizes the roll-outs and produce good results even for cases where without noise the roll-out trajectories diverge towards infinity. We can also see from figure 3.9 that the model can learn to use

	ϕ $[rad]$	$\dot{\phi}$ $[rad/s]$	${}_B g$ $[\%]$	${}_W r_{WB,z}$ $[m]$	${}_B v_{WB}$ $[m/s]$	${}_B \omega_{WB}$ $[rad/s]$	*Average*
walk	0.019	0.796	0.008	0.030	0.022	0.078	**0.30**
random	0.018	0.982	0.020	0.020	0.027	0.151	**0.37**

Table 3.2: Mean absolute error of one-step predictions per component of the observations for the random and walking policies. Each row is obtained by training and validating on the corresponding policy. The last column represents a score obtained by all previous columns with respective units. It has no physical meaning, but allows to summarise the performance in a single number. The stochastic model and a noise level of 0.5σ are used, averaged over 45 validation trajectories

	ϕ $[rad]$	$\dot{\phi}$ $[rad/s]$	${}_B g$ $[\%]$	${}_W r_{WB,z}$ $[m]$	${}_B v_{WB}$ $[m/s]$	${}_B \omega_{WB}$ $[rad/s]$	*Average*
walk	0.057	0.982	0.020	0.060	0.039	0.176	**0.39**
random	0.410	2.084	0.312	0.158	0.253	0.945	**1.02**

Table 3.3: Mean absolute error of roll-outs per component of the observations for the random and walking policies. Each row is obtained by training and validating on the corresponding policy. The stochastic model and a noise level of 0.5σ are used. Averaged over 45 validation trajectories. Last column as in table 3.2

the window of inputs to filter out the noise and make predictions beyond the noise level.

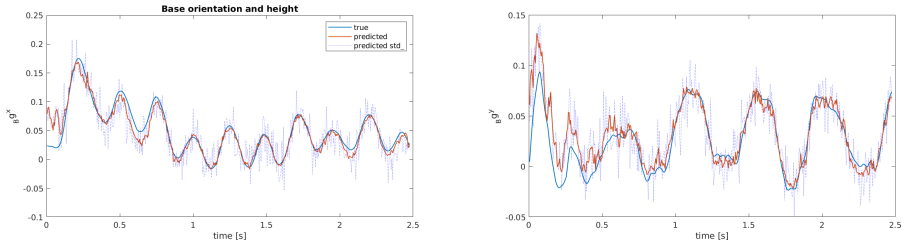


Figure 3.9: One-step prediction for a validation trajectory. The noise level represents the amount of noise added to the inputs during training.

One possible interpretation of this phenomenon is that by adding noise to the observations, the difference between o_k and o_{k+1} becomes larger so the simple prediction $\hat{o}_{k+1} \approx o_k$ is not good enough any more. The model is forced to use the windows and learn a more complex function.
We can compare both the training loss and the roll-out mean absolute error for different levels of noise. Since the inputs and outputs are normalized, the amount of noise is calculated relative to the standard deviation of each component of the observation. Figure 3.10 shows that up a noise level of 0.5σ the training loss increases, while the roll-out error decreases.

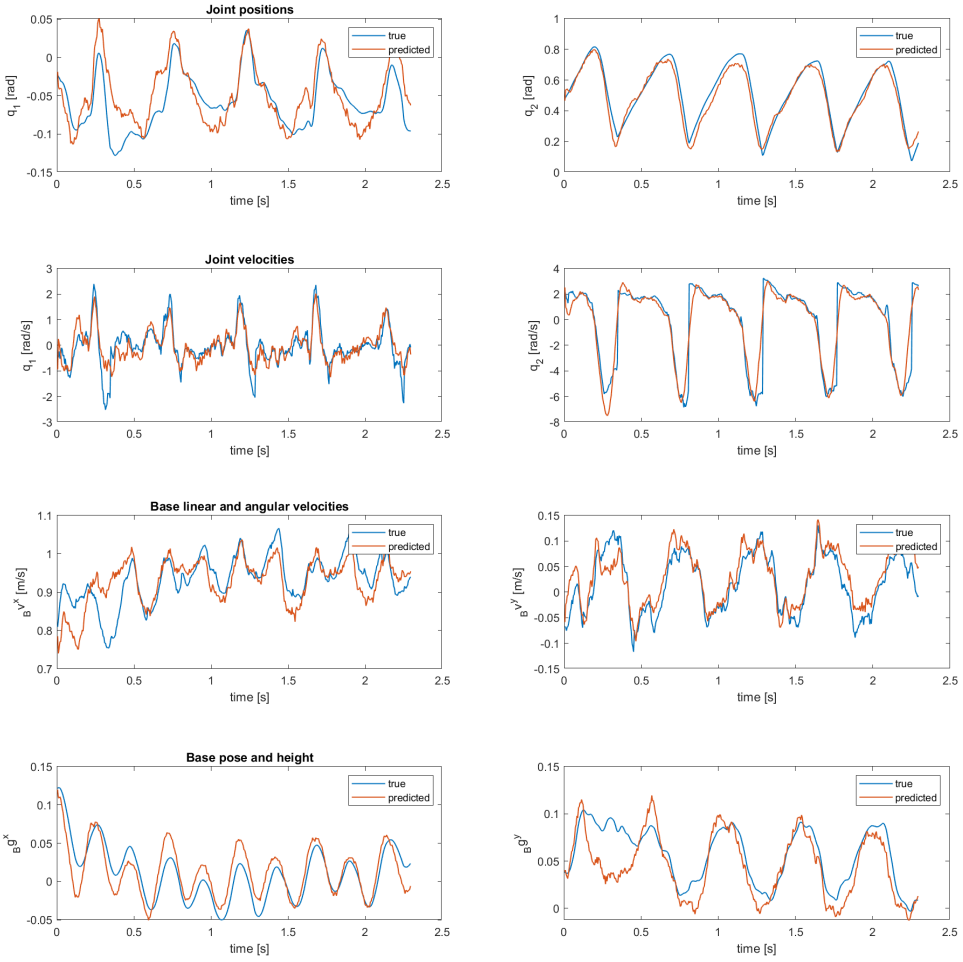
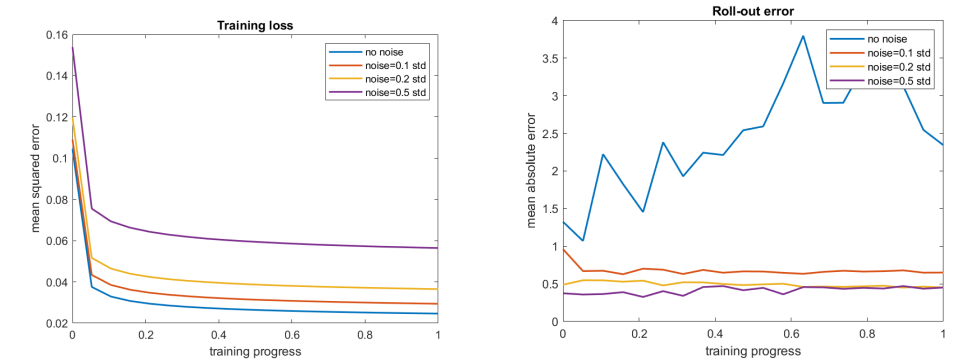


Figure 3.8: Roll-out predictions of a validation trajectory for selected components (same as in figure 3.7) of the observation. Stochastic model with 0.5 σ noise, walking policy.



(a) Training loss.

(b) Roll-out mean absolute error

Figure 3.10: Comparison of the training loss and roll-out error for noise levels of 0, 0.1, 0.2 & 0.5 σ

	walk one-step	random one-step	walk roll-out	random roll-out
mse	0.20	0.29	0.82	1.81
stochastic	0.30	0.37	0.39	1.02

Table 3.4: Comparison of average (as in table 3.2) one-step and roll-out mean absolute errors for the simple and stochastic models, with a noise level of 0.5σ and averaged over 45 validation trajectories

3.5.4 Mean Squared Error versus Log-Likelihood

As described in the previous section, the same model can be used with a simple mean squared error or by adding an extra layer, it can predict a diagonal Gaussian distribution used with negative log likelihood loss. An immediate advantage of the second approach is that the model predicts its own uncertainty. This can be useful in a variety of scenarios. For example, if the uncertainty becomes too high, the trajectory can be discarded from the set used to train a policy. Figure 3.11 provides an example of a prediction, where the model struggles to match the target, but the uncertainty correctly increases.

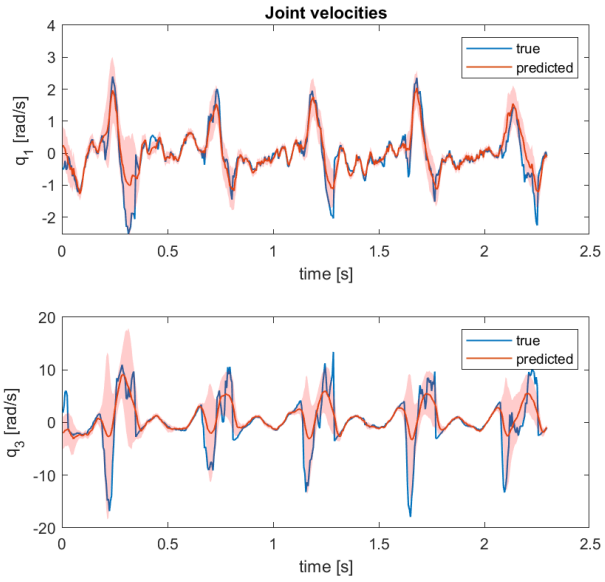


Figure 3.11: One-step prediction on validation trajectory with predicted covariance in light red. The covariance increases where the prediction is wrong.

In terms of performance, table 3.4 summarizes the predictive power for both one-step predictions and roll-outs on the validation set. Once again, we see that one-step and roll-out performance are not correlated. While the simple mean squared error is slightly better at one-step predictions, it is highly outperformed by the stochastic approach in roll-outs.

3.5.5 Lack of Generalization Between Policies

As a verification of the quality of the learned model. We can test it on a policy it has not seen in the training data. Unfortunately, the results are not great. Figure 3.12 shows what happens when the model is trained on the walking policy , but has to perform a roll-out with actions coming from the bigait one. It seems that the model is not learning the true dynamics in the physical sense, but finds short-cuts that are conditioned on the policy. One simple solution is to train the model on a variety of policies providing richer data. Even in that case there is no guarantee that the model is learning a more general function instead of identifying the policy and choosing one of multiple conditioned sub-models. Furthermore, we can see from 3.3 that a random policy causes much larger errors. This strengthens the hypothesis that the model relies on cyclic patterns in the walking policy.

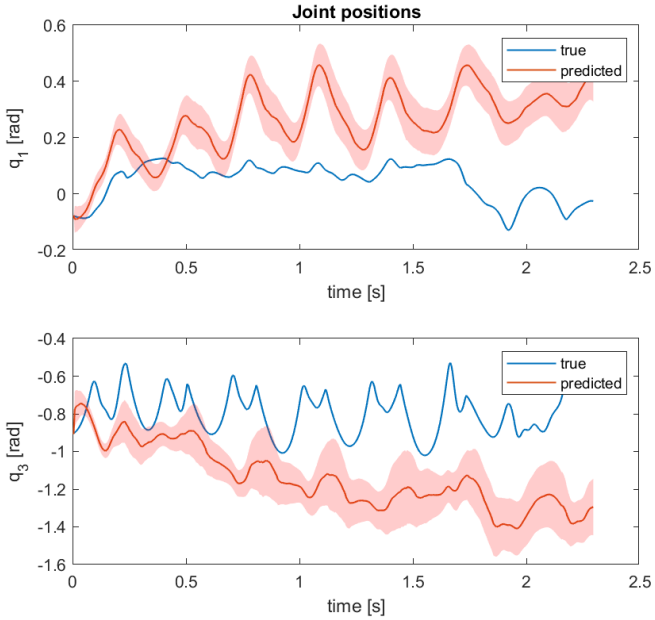


Figure 3.12: Roll-out predictions for the bigait policy. The model was trained on the walking policy.The stochastic model and a noise level of 0.5σ are used

3.5.6 Training on Combined Datasets

One way to force the network to learn the true dynamics of the environment is to have a richer dataset. By having enough of different policies, gaits and behaviours in the dataset, it should be easier for the network to learn the actual physical dynamics than to find other ways of making good predictions. Effectively, we remove the potential short-cuts the model was using in the case of repetitive or cyclic data. For example, in the case of the walking policy, the base of the robot is always moving forward with approximately the same velocity. The model will then predict this velocity independently of any inputs. It is therefore useful to have data where the robot is moving in different directions with various velocities. We can train our model on a combined dataset containing trajectories from all three simulator policies. We then test it on each individual policy. The data collected on the real robot is even richer because it contains various gaits and varying base velocity

commands. As opposed to the simulator policies, the gaits used on the real robot are able to rotate as well as walk sideways and backwards. Table 4.2 provides results of such training on the combined simulator policies, while table 4.3 contains results for the real robot. These results are worse than in table 3.3. It seems that under one-step training, the model is not able to correctly learn the dynamics of multiple policies.Increasing the model size and optimizing hyper-parameters did not significantly improve these results. Appendix A.1 contains plots of roll-out predictions for the real robot. From these plots it is clear that the model struggles with the complex real world data.

	ϕ $[rad]$	$\dot{\phi}$ $[rad/s]$	${}_B g$ $[\%]$	${}_W r_{WB,z}$ $[m]$	${}_B v_{WB}$ $[m/s]$	${}_B \omega_{WB}$ $[rad/s]$	*Average*
walk	0.116	1.404	0.039	0.127	0.086	0.281	**0.574**
random	0.352	2.165	0.199	0.619	0.390	0.786	**1.015**
bigait	0.204	1.688	0.188	0.285	0.290	0.756	**0.779**

Table 3.5: Mean absolute error of roll-out predictions per component of the observations for the random, bigait and walking policies, trained on the combined dataset, tested on each individual policy and averaged over 50k samples, last column as in table 3.2.

	ϕ $[rad]$	$\dot{\phi}$ $[rad/s]$	${}_B g$ $[\%]$	${}_W r_{WB,z}$ $[m]$	${}_B v_{WB}$ $[m/s]$	${}_B \omega_{WB}$ $[rad/s]$	*Average*
stand	0.218	0.713	0.083	0.295	0.160	0.513	**0.396**
crawl	0.154	0.931	0.010	0.235	0.155	0.535	**0.446**
trot	0.068	0.834	0.004	0.268	0.155	0.160	**0.347**
flying-trot	0.096	0.974	0.010	0.626	0.419	0.208	**0.436**

Table 3.6: Mean absolute error of roll-out predictions per component of the observations for different gaits of the real robot, trained on the full dataset, tested on each individual gait and averaged over 12k samples, last column as in table 3.2.

Chapter 4

Masked Sequence Training

4.1 Motivation

We have seen in the previous chapter that the one-step training presents some severe challenges, which can be summarized as follows:

- Necessity to add a high level of noise, reducing the potential precision.
- Lack of generalization between policies.
- Difficulties with a random policy

All these problems are related to the roll-outs. In the previous approach, it is hard to control or even predict the roll-out performance because the model is not trained for it. This second approach is motivated by the idea that a one-step training and roll-outs are not correlated enough. We will show how the training can be done directly on roll-outs.

4.2 Description

The idea for this approach comes from conditional sequence generation, where a sequence is generated conditioned on an extra input. For example the WaveNet [7] model can generate speech conditioned on the speaker identity (i.e different voices). In this particular case, the identity of the speaker is provided at each step as an extra input.
We can view our problem set-up as a task of generating a sequence of observations with actions as input and conditioned on the initial state. Replicating the WaveNet approach, we could provide the initial state to all inputs (replace o_k with o_0). Intuitively, it makes more sense to only give the initial state to the first input (remove o_k for $k > 0$). Because a temporal convolutional layer needs all of the time steps to have the same dimension, we can not simply remove part of the input. We can, however, mask them with zeros, effectively removing the information.
Finally, we provide as input the full trajectory of actions together with the initial observation padded with a long sequence of zeros. As output target we still expect our model to predict a complete trajectory of observations. Note that since we directly want to train for long-term predictions we do not cut the sequence into small windows. The inputs and outputs now have as many time steps as there are in a trajectory (500 in the following experiments). It is important that even the last prediction has the initial observation in its receptive field, as such we need to make use of dilated convolutions and use up to 7 blocks in the TCN with dilations of [1, 2, 4, 8, 16, 32, 64] for a sequence of 500 steps. Figure 4.1 shows this set-up that we call masked sequence training.

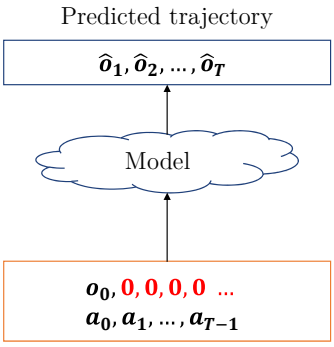


Figure 4.1: Set-up of the masked sequence training

4.2.1 On Policy Predictions

As discussed before, in a model-based reinforcement learning set-up we would like to have the model interact with a policy. This means that we do not know the trajectory of actions beforehand. We need to provided an intermediate observation to the policy in order to receive the corresponding action. This is still possible in this proposed approach. Due to the causality of the network, $\hat{o}_k$ does not depend on o_{k+1}, a_{k+1}. As such we can use the following strategy:

1. Start with a trajectory only containing o_0, a_0 and padded with zeros as inputs. Make a prediction.
2. Provide the first prediction $\hat{o}_1$ to the policy. Obtain a_1.
3. Insert a_1 in the input. Make a prediction.
4. Provide $\hat{o}_2$ to the policy. Obtain a_2.
5. Continue until the end of the trajectory.

Figure 4.2 shows this procedure graphically. One potential drawback of this approach is that a lot of extra computations are executed when predicting the beginning of the trajectory. This could be avoided by having a dynamic graph, where the dimension of the time dimension in the temporal convolution is allowed to change.

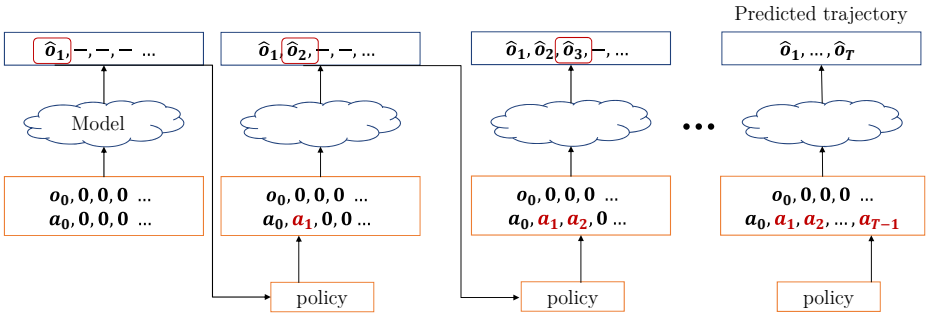


Figure 4.2: On policy prediction strategy using masked sequence training.

4.3 Mixture Density Network

Since this second training approach is more complex than the previous one. It requires a more capable model. We propose to use a mixture density network as an extension to the previous model. The main part of the model remains a TCN network, but this time it outputs M modes of both mean $\hat{o}_k$ and covariances $\hat{\sigma}_k$ (if the stochastic version is used) along with M probabilities π_m. A weighted sum is then applied for the final prediction. The output is then

$$\tilde{o}_k = \sum_{m=1}^{M} \pi_m \hat{o}_k$$

in the stochastic case a simplification is used. The output is

$$\tilde{o}_k = \sum_{m=1}^{M} \mathcal{N}(\pi_m \hat{o}_k, \pi_m \hat{\sigma}_k)$$

instead of:

$$\tilde{o}_k = \sum_{m=1}^{M} \pi_m \mathcal{N}(\hat{o}_k, \hat{\sigma}_k)$$

The number of modes is selected as the number of possible contact configurations under standard walking. Since each of the 4 feet of the robot is either in contact or not, we have

$$M = 2^4 = 16$$

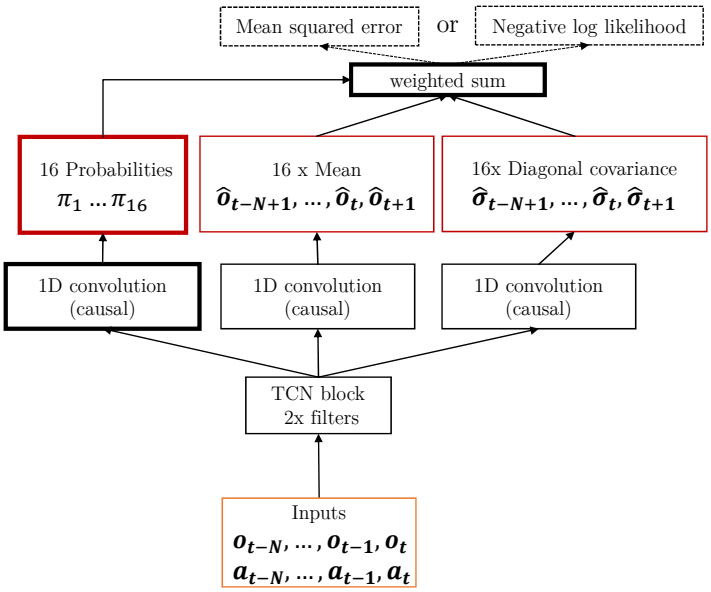
Clearly this model will have more parameters. It also requires a larger number of filters in the TCN network. In the following experiments the number of filters was doubled, effectively multiplying the number of parameters by a factor of 5 for the complete network. Adding even more filters might lead to better performance, but this limit was dictated by the memory (6GB) of the available GPU. Figure 4.3 summarises this model.

4.4 Results

Results in this section are no longer split into one-step and roll-out predictions. In this approach all predictions are equivalent to roll-outs. Importantly, there is no need to add noise to the inputs under this training scheme. Plots in figure 4.4 show some predictions of a mixture density model with a mean squared error. For a clear performance comparison, we can compare table 4.1,which summarises mean absolute errors along validation trajectories to table 3.3. We observe a clear increase in performance, especially on the random policy. It might seem like a

	ϕ $[rad]$	$\dot{\phi}$ $[rad/s]$	${}_B g$ $[\%]$	${}_W r_{WB,z}$ $[m]$	${}_B v_{WB}$ $[m/s]$	${}_B \omega_{WB}$ $[rad/s]$	*Average*
walk	0.045	0.686	0.021	0.050	0.038	0.130	**0.275**
random	0.110	1.289	0.053	0.089	0.111	0.396	**0.544**

Table 4.1: Mean absolute error of roll-outs per component of the observations for the random and walking policies. Each row is obtained by training and validating on the corresponding policy, without noise and averaging over 45 validation trajectories, last column as in table 3.2.

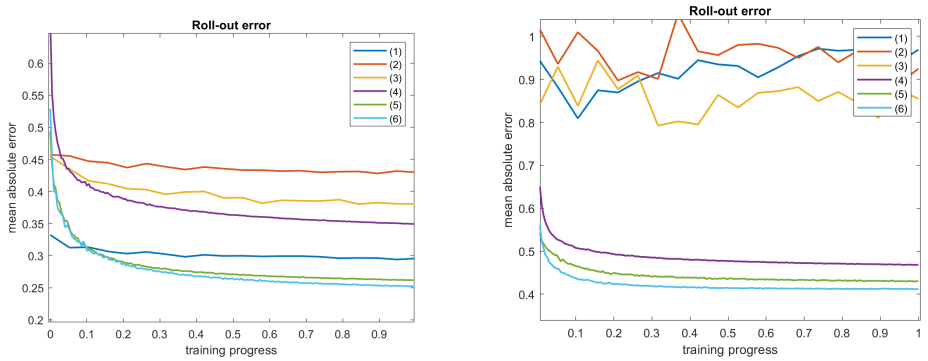


1

Figure 4.3: Mixture density model with an increased TCN

comparison with the previous results is unfair given that the model now has 4 times more parameters. In order to amend this difference, we can train a larger TCN with the same number of parameters as the mixture density model. Finally we compare the performance of the following model and training scheme combinations in figure 4.5:

1. TCN with $\approx 70k$ parameters, one-step training.
2. TCN with $\approx 350k$ parameters, one-step training.
3. Mixture density model with $\approx 350k$ parameters, one-step training.
4. TCN with $\approx 70k$ parameters, masked sequence training.
5. TCN with $\approx 350k$ parameters, masked sequence training.
6. Mixture density model with $\approx 350k$ parameters, masked sequence training.



(a) Walking policy.

(b) Random policy

Figure 4.5: Comparison of the roll-out mean absolute error for the described configurations.

As expected only increasing the model complexity in one-step training does not

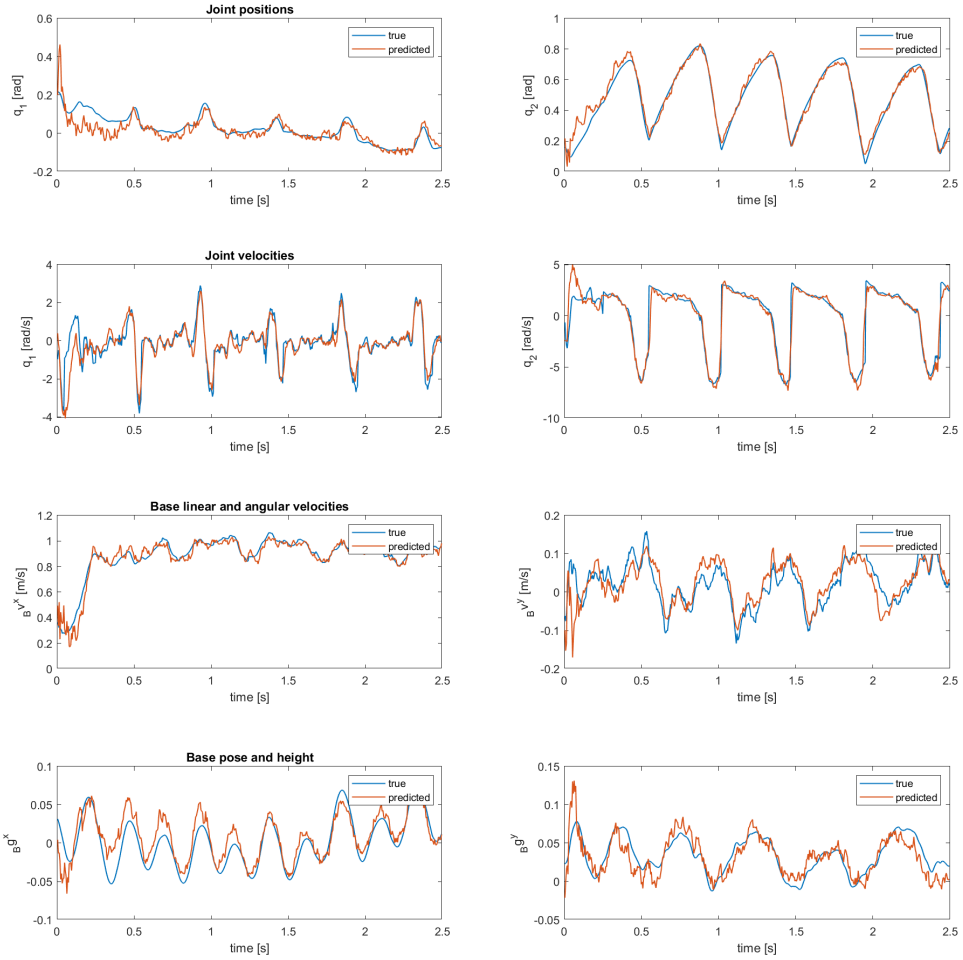


Figure 4.4: Prediction of validation trajectories of the mixture density model trained with masked sequences. Walking policy, no noise

improve roll-out performance and actually makes it worse. Switching to masked sequence training provides an immediate improvement for the simple and increased TCN models. Finally the mixture density model with masked sequence training produces the best performance in both settings.

4.4.1 Mean Squared Error versus Log-Likelihood

Surprisingly, when using a mixture density model with masked training the stochastic version of the model doesn't seem to produce better results. This might be due to the added complexity of predicting the diagonal covariance or because of the simplification described in 4.3. Nevertheless, the simpler mean squared error seems to produce slightly better results. The stochastic model still has the advantage of predicting its own uncertainty. We can verify that the uncertainty is learned correctly under this type of training. Figure 4.6

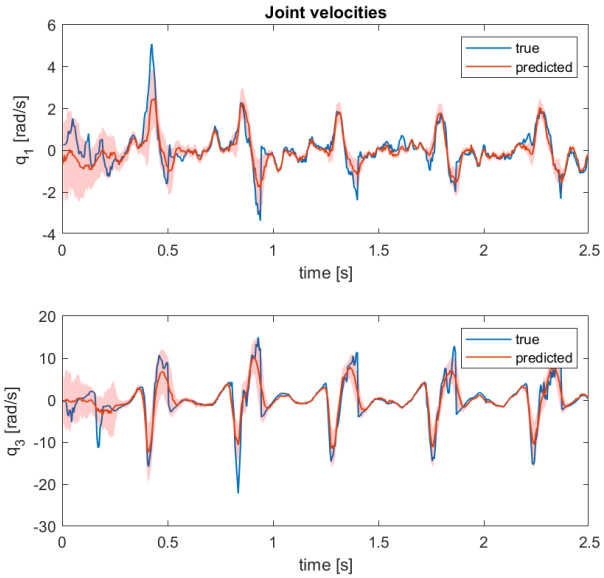


Figure 4.6: Predictions with uncertainty of the mixture density network under masked sequence training.

4.4.2 Conditioning on the Initial Observation

In the previous chapter we hypothesised that the model requires a history of past observations and actions to correctly predict the future. It might seem strange that we are only giving o_0 instead of a window $o_{0:N}$ of initial observations. Actually, the implementation allows to select the size of the window of initial observations that remains unmasked. Empirically, it seems that o_0 is enough and adding a window does not help the model. Intuitively, it can be understood by studying the reason we need a history of inputs. We are trying to include delays and other actuator effects related to springs into the model, these clearly require a history of commands, but this only concern the transition from actions to the true physical torques in the joints. Once we have these torques a single observation is enough to predict the next transition. As such, o_0 combined with a full history of actions should be enough for the model.

This being said, a strange effect requires investigation: the model is able to make predictions without receiving o_0. It is able to predict a full trajectory of observations only from the actions. The errors are larger than when it has access to o_0, but still much better than random. One possible reason is that instead of learning the dynamics, the model is able to get the observations directly from actions by inverting the function hidden inside the policy. This is most probably not the case because this effect is still present with the random policy. A second option is that the initial state is always approximately the same. The model can simply learn this particular initial state from the training data. Finally, it might be the case that after some time, the state of the robot only weakly depends on the initial state. This is especially true with the random policy. With each random transitions the dependency on the initial state is decreased.

Most probably, a combination of these factors is at play. Further investigation would be needed.

4.4.3 Training on Combined Datasets

As in the previous chapter, we can train our model on the combined dataset. We then test it on each individual policy. Table 4.2 provides results of such training on the combined simulator policies, while table 4.3 contains results for the real robot. Apendix A.2 contains plots of predictions for the real robot. Clearly, both the tables and plots show a large increase in predictive power compared to the previous approach.

	ϕ $[rad]$	$\dot{\phi}$ $[rad/s]$	${}_B g$ $[\%]$	${}_W r_{WB,z}$ $[m]$	${}_B v_{WB}$ $[m/s]$	${}_B \omega_{WB}$ $[rad/s]$	*Average*
walk	0.048	0.746	0.014	0.066	0.048	0.146	**0.299**
random	0.092	1.252	0.036	0.101	0.094	0.306	**0.513**
bigait	0.035	0.612	0.013	0.057	0.044	0.125	**0.244**

Table 4.2: Mean absolute error of masked sequence predictions per component of the observations for the random, bigait and walking policies. Trained on the combined dataset, tested on each individual policy and averaged over 50k samples. Last column as in table 3.2.

	ϕ $[rad]$	$\dot{\phi}$ $[rad/s]$	${}_B g$ $[\%]$	${}_W r_{WB,z}$ $[m]$	${}_B v_{WB}$ $[m/s]$	${}_B \omega_{WB}$ $[rad/s]$	*Average*
stand	0.029	0.194	0.011	0.041	0.029	0.088	**0.090**
crawl	0.022	0.210	0.004	0.137	0.111	0.048	**0.097**
trot	0.022	0.215	0.004	0.378	0.245	0.054	**0.111**
flying-trot	0.028	0.276	0.004	0.671	0.431	0.093	**0.155**

Table 4.3: Mean absolute error of masked sequence predictions per component of the observations for different gaits of the real robot. Trained on the full dataset, tested on each individual gait and averaged over 12k samples. Last column as in table 3.2.

Chapter 5

Discussion

5.1 Two Training Approaches

We have seen two different approaches for training the network. While the first might seem more standard and intuitive, it has the disadvantage of having different training (one-step) and test (roll-out) scenarios. The network is not directly trained on the task it is supposed to learn. We have seen that this leads to inconsistent performance between the two types of predictions. The second approach tries to solve this issue by directly training on the test scenario. The main idea is to avoid giving more information during training than it would get during a roll-out. Masking all other observations in order to provide only the initial one might not be the best implementation of this idea, but it achieves its purpose and actually improves the performance of the model. It heavily outperforms the previous approach in the case of a random policy and on the real world data and doesn't require the addition of input noise. It seems to be a promising direction and would benefit from further investigation.

5.2 Mixture Density Network Versus Single Mode

In the previous chapter, we have introduced mixture density networks as a more powerful model. We have seen that as expected, in the case of one-step training it reduces the training loss, but increases roll-out errors. In that scenario a more powerful model seems to be detrimental. In masked sequence training however, the task is more challenging and a more powerful model helps to reduce the roll-out error. In order to understand the benefits of the specific representation of a mixture density network, we have compared it to an increased TCN with the same number of parameters, but predicting a single mode. We have noticed that the mixture density network performs slightly better under all scenarios.

5.3 Stochastic versus Non-Stochastic Model

We have introduced a stochastic model that predicts a state dependant diagonal covariance. It has the advantage of predicting its own uncertainty. It can also improve the quality of the predictions by treating complicated samples differently and by using a large covariance on components of the observation that are harder to predict, while keeping a more precise prediction for simpler components. Note that this last advantage is not as relevant for normalized inputs. Under one-step training we have seen that the stochastic model outperforms the non-stochastic ver-

sion in terms of roll-out error. Under the masked training we do not observe this advantage any more. Actually switching to stochastic predictions slightly reduces the performance. The reason for this remains unclear, it could be due to the increased complexity of predicting the uncertainty and might be solved by increasing the size of the network even further. Finally, it would be necessary to check that the simplification made in the stochastic mixture density network is not having a negative influence on the performance.

5.4 Training on Richer Datasets

As discussed before, training on a richer dataset seems to be a good solution to the problem of lack of generalization. This was the main motivation for collecting a dataset on the real robot, where multiple gaits with more control were available. Unfortunately, new problems emerged in this data. Most notably, the estimation of the base height ${}_{W}r_{WB,z}$ is not reliable. Even after corrections, the estimate is not precise and by visualizing the saved data we can observe that the feet of the robot are slightly above or below the ground. This can have a strong detrimental effect on the quality of the learning process, because the contacts can not be inferred precisely. It may even force the network to find new short-cuts to make predictions. Nevertheless, experiments were conducted on this dataset and we have observed that the model was able to make decent predictions for all of the recorded gaits.

5.5 Potential Future Work

In order to continue the presented work, the first step would be to implement a complete model-based reinforcement learning algorithm in order to understand the capabilities of the proposed model in its real use-case. Furthermore, future developments can include the following investigations:

- A TCN can be used for other types of predictions where a temporal component is important. These can include, contact predictions, actuator models, disturbance estimation, fall detection etc.
- The model capabilities can be improved by exploring more complex architectures. An example would be to use a generative adversarial network (GAN) where the discriminator can tell if a sequence comes from the model or the true dataset. This could make the model learn physical consistency (i.e. the dynamics) without forcing it reproduce the training data.
- Other types of models can include graphical neural networks or first-order-principles-based network [11], which include an explicit structure in the learned function.
- Physical consistency can be enforced explicitly via a hand-crafted loss in the model.
- More structure can be given to the model, by combining a learned function with standard kinematics or dynamics.

Chapter 6

Conclusion

In its entirety this work has shown that a neural network can learn to predict end-to-end the evolution of a complex robot such as ANYmal, but it is extremely hard to control or understand the learning process. This is of course a common characteristic of end-to-end learning approaches. In our case there can be different patterns present in the dataset and there is no guarantee that the learned function is related to the true dynamics in the physical sense. It is important to note that it might not be necessary to learn the true dynamics. As long as the learned function generalises to new trajectories under a similar policy, the model can be used in the Dyna/World models set-up. In the end, it is unclear if any of the proposed models perform well enough for this task, but a mixture density network trained with masked sequences seems to be the best candidate.
An end-to-end approach might not be the best solution. We have a lot of knowledge about the structure of the problem, but we are not enforcing it in any way. Solutions can include, a re-parametrisation of the predictions, combining kinematics with a learned function or an other type of network.
We have also seen that both in terms of amount of data and computational complexity this sort of model can be learned directly on the real robot. Meanwhile, when training in a simulator, the only advantage could be faster training. Since the model inference is not substantially faster than the physical simulator, this doesn't seem to be case. As such, one approach could be to pre-train a policy under model-free reinforcement learning in the simulator, then improve it by running a Dyna type algorithm on the real robot.

Bibliography

[1] D. Ha and J. Schmidhuber, "World Models," *arXiv e-prints*, p. arXiv:1803.10122, Mar 2018.

[2] L. Kaiser, M. Babaeizadeh, P. Milos, B. Osinski, R. H. Campbell, K. Czechowski, D. Erhan, C. Finn, P. Kozakowski, and S. Levine, "Model-Based Reinforcement Learning for Atari," *arXiv e-prints*, p. arXiv:1903.00374, Mar 2019.

[3] J. Hwangbo, J. Lee, A. Dosovitskiy, D. Bellicoso, V. Tsounis, V. Koltun, and M. Hutter, "Learning agile and dynamic motor skills for legged robots," *arXiv e-prints*, p. arXiv:1901.08652, Jan 2019.

[4] R. S. Sutton, "Dyna, an Integrated Architecture for Learning, Planning, and Reacting," in *Working Notes of the 1991 AAAI Spring Symposium*, 1991, pp. 151–155.

[5] S. Bengio, O. Vinyals, N. Jaitly, and N. Shazeer, "Scheduled Sampling for Sequence Prediction with Recurrent Neural Networks," *arXiv e-prints*, p. arXiv:1506.03099, Jun 2015.

[6] "Wayve dreaming about driving," https://wayve.ai/blog/dreaming-about-driving-imagination-rl, accessed: 17-06-2019.

[7] A. van den Oord, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. Senior, and K. Kavukcuoglu, "WaveNet: A Generative Model for Raw Audio," *arXiv e-prints*, p. arXiv:1609.03499, Sep 2016.

[8] S. Bai, J. Zico Kolter, and V. Koltun, "An Empirical Evaluation of Generic Convolutional and Recurrent Networks for Sequence Modeling," *arXiv e-prints*, p. arXiv:1803.01271, Mar 2018.

[9] M. contributors, "Keras temporal convolutional network," https://github.com/philipperemy/keras-tcn, 2013.

[10] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[11] F. Díaz Ledezma and S. Haddadin, "First-order-principles-based constructive network topologies: An application to robot inverse dynamics," in *2017 IEEE-RAS 17th International Conference on Humanoid Robotics (Humanoids)*, Nov 2017, pp. 438–445.

Appendix A

A.1 One-step Training

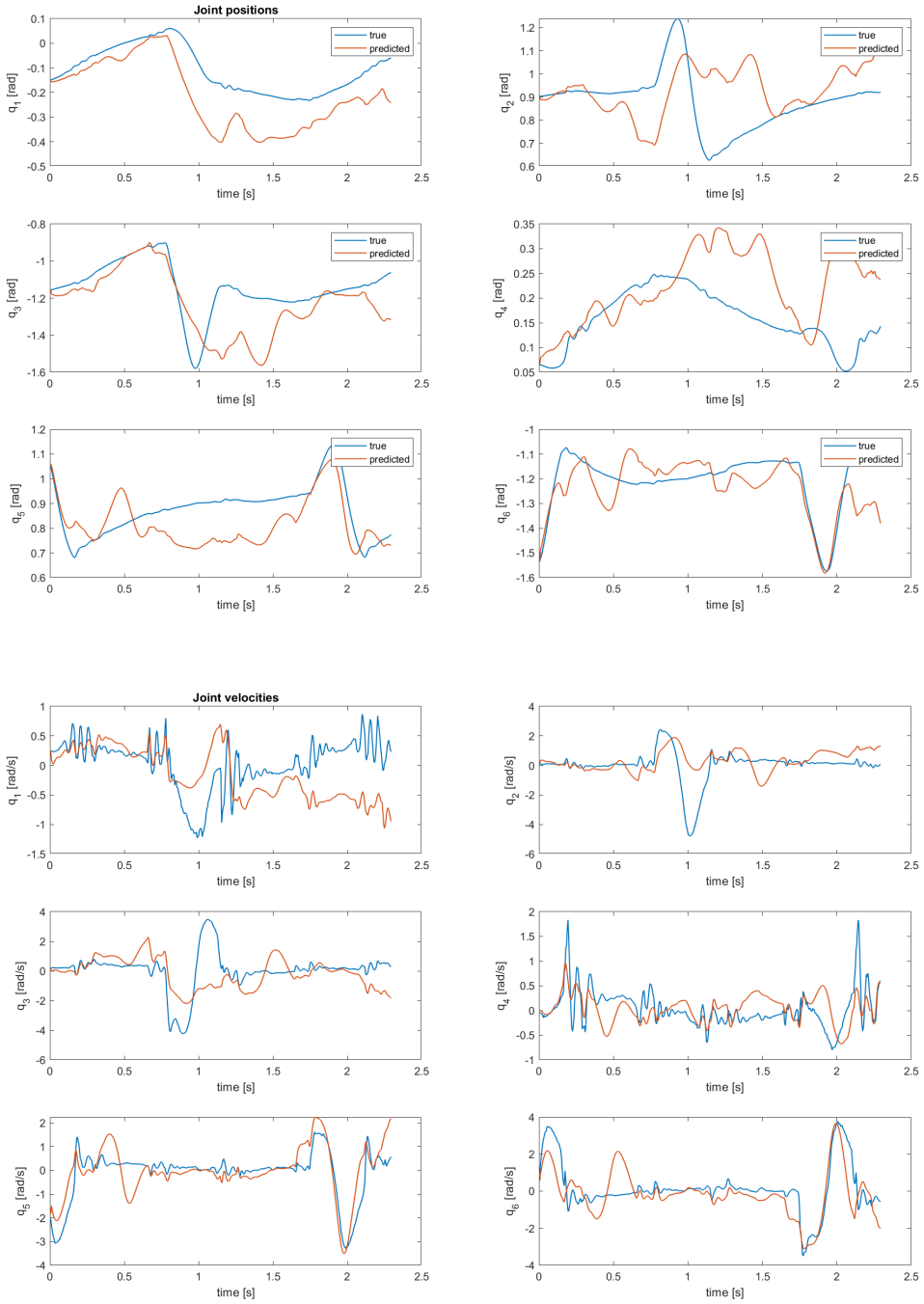


Figure A.1: Roll-out predictions on a validation trajectory of the full real robot dataset, under one-step training (1/2).

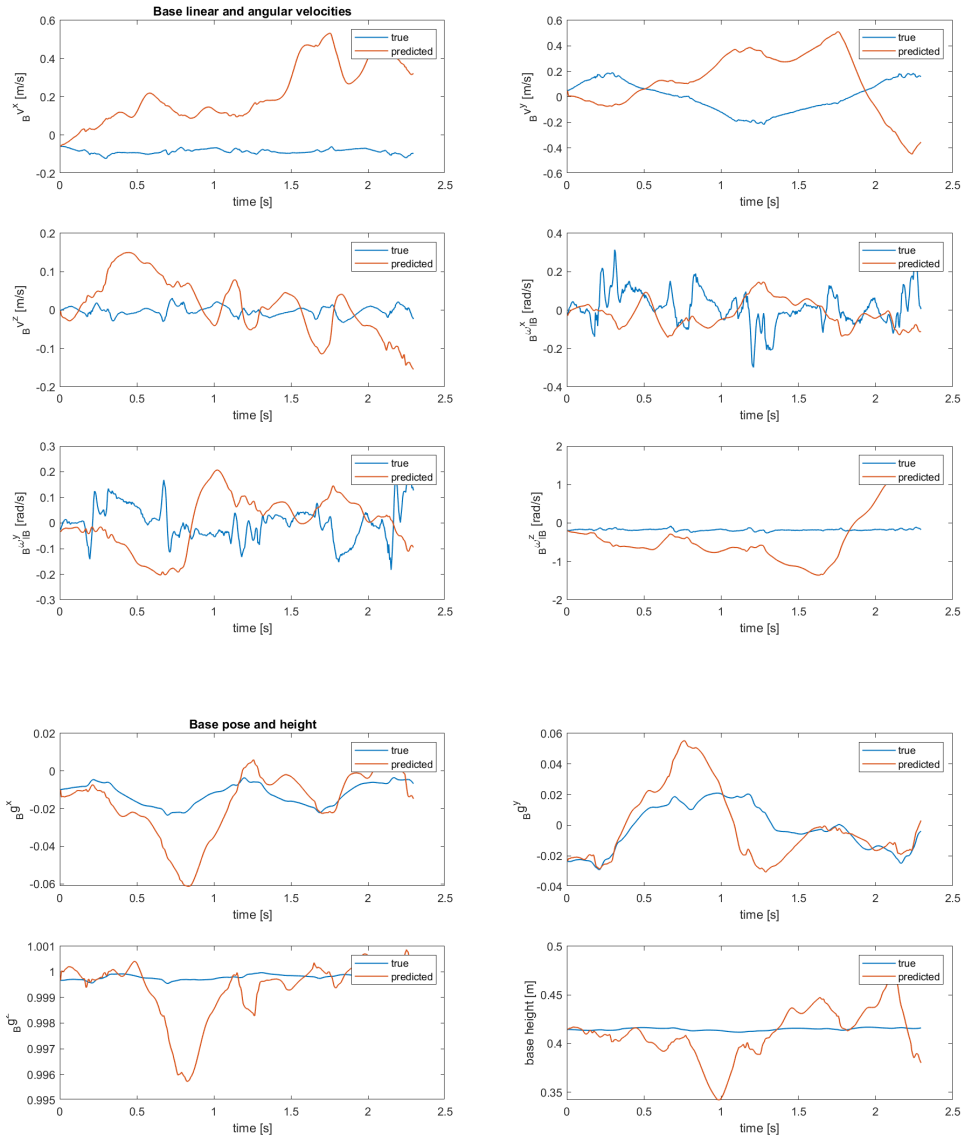


Figure A.2: Roll-out predictions on a validation trajectory of the full real robot dataset, under one-step training (2/2).

A.2 Masked Sequence Training

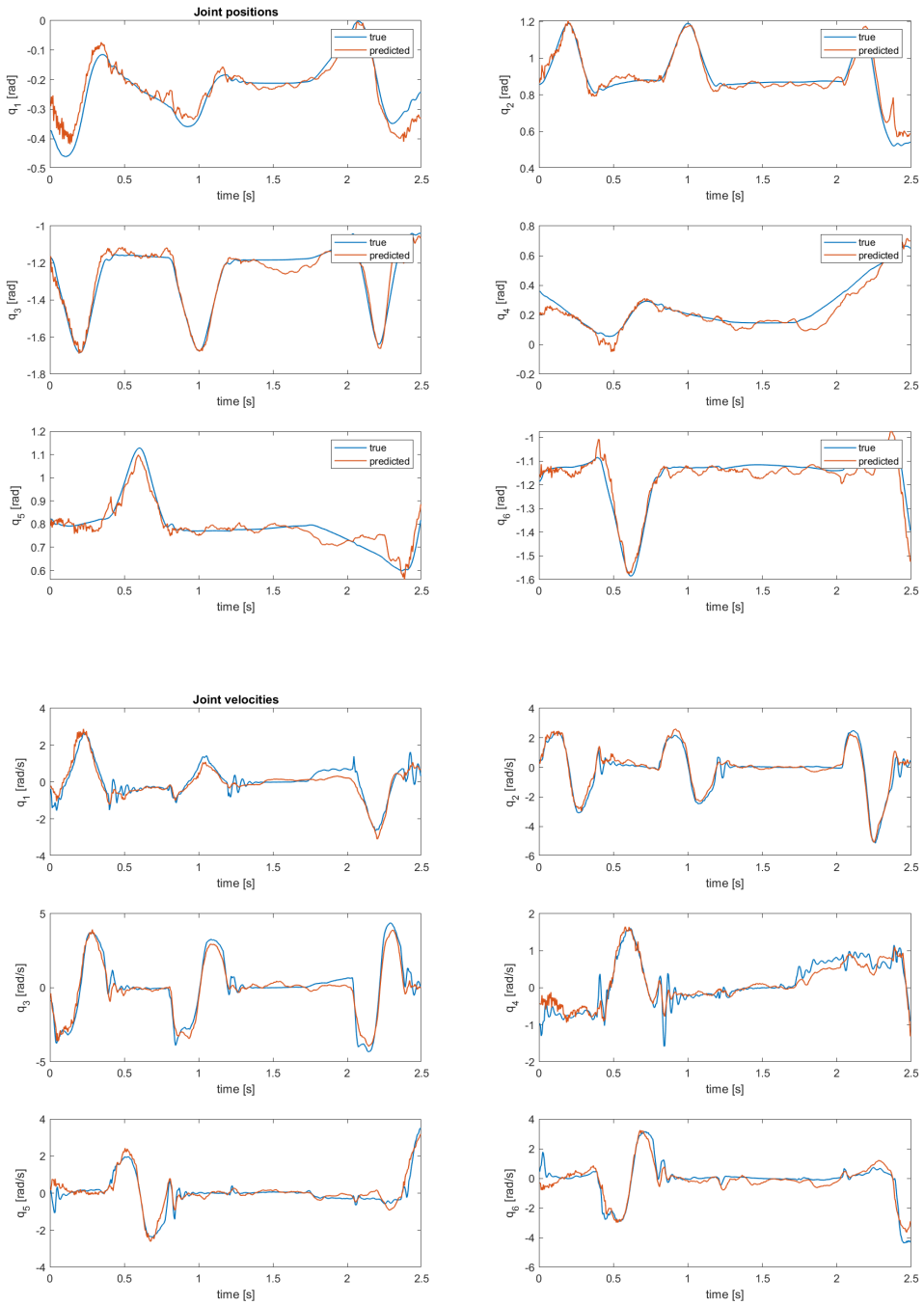


Figure A.3: Predictions on a validation trajectory of the full real robot dataset, under masked sequence training (1/2).

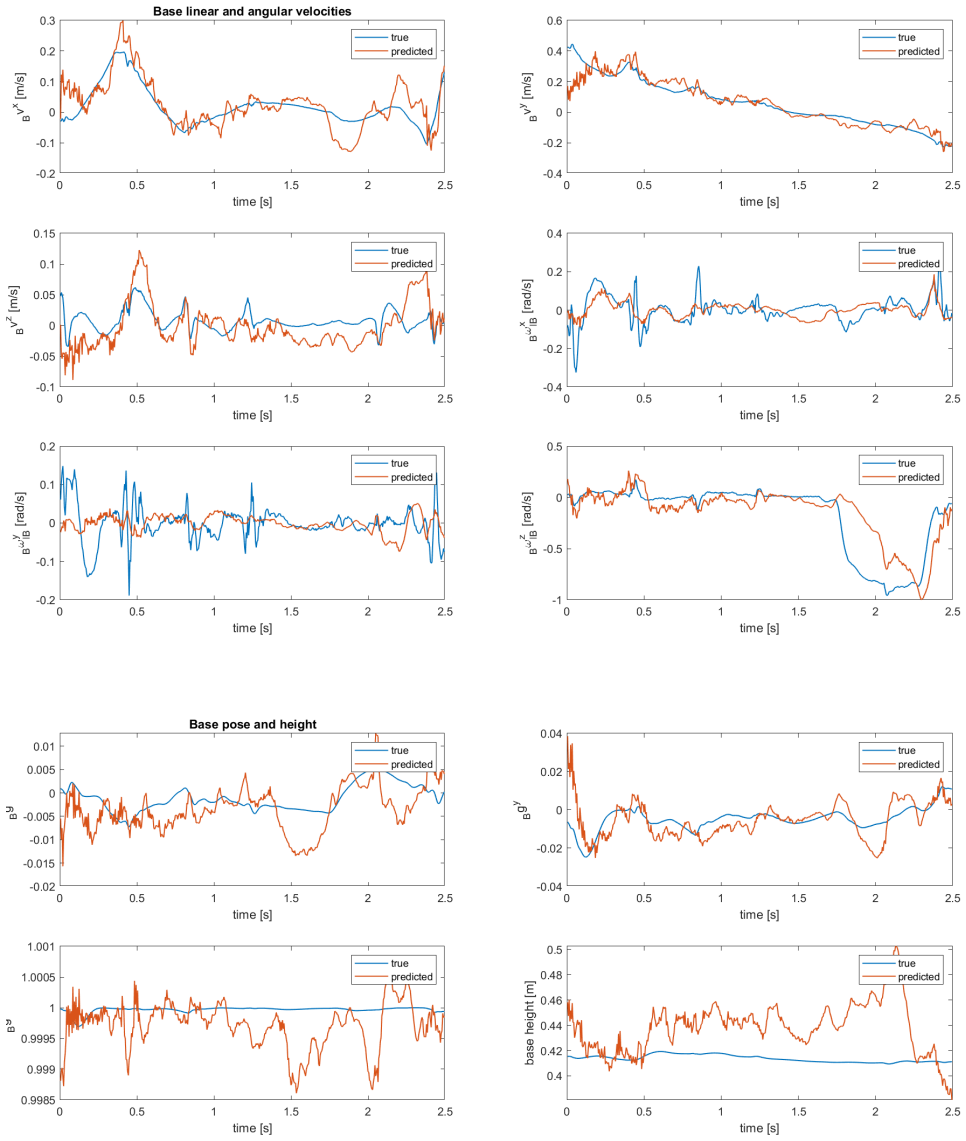


Figure A.4: Predictions on a validation trajectory of the full real robot dataset, under masked sequence training (2/2).