Realeasy: Real-Time capable Simulation to Reality Domain Adaptation

Dürr, Alexander; Neric, Liam; Krueger, Volker; Topp, Elin Anna

Published in:
2021 IEEE 17th International Conference on Automation Science and Engineering (CASE)

DOI:
10.1109/CASE49439.2021.9551626

2021

Document Version:
Peer reviewed version (aka post-print)

Realeasy: Real-Time capable Simulation to Reality Domain Adaptation*

Alexander Dürr[1], Liam Neric[1], Volker Krueger[1], and Elin A. Topp[1]

***Abstract*— We address the problem of insufficient quality of robot simulators to produce precise sensor readings for joint positions, velocities and torques. Realistic simulations of sensor readings are particularly important for real time robot control laws and for data intensive Reinforcement Learning of robot movements in simulation. We systematically construct two architectures based on Long Short-Term Memory to model the difference between simulated and real sensor readings for online and offline application. Our solution is easy to integrate into existing Robot Operating System frameworks and its formulation is neither robot nor task specific. We demonstrate robust behavior and transferability of the learned model between individual Franka Emika Panda robots. Our experiments show a reduction in torque mean squared error of at least one order of magnitude. The collected data set, the plug-and-play *Realeasy* model for the Panda robot and a reproducible real-time docker setup are shared alongside the code.[2]**

I. Introduction

For many interesting robot control problems in complex environments it is necessary to have a fast and reliable controller. Model Predictive Controllers (MPC) estimate the effect of a control action in such environments. However, the non-linear model of the robot dynamics is usually approximated by a linear or quadratic model to allow calculations in real-time. This model approximation creates an error in the prediction. The resulting residual can cause the controller to select sub-optimal control actions.

In Reinforcement Learning (RL) we face a related problem: To find a good policy, a lot of exploration of the environment by the learning agent is needed. Ideally this is handled in *simulation*, as it is fast and safe. However, when the learned policy is deployed to the *real environment*, the real robot state transitions usually look different from the simulated state transitions despite being in the same situation. This causes the policy to pick a possibly sub-optimal action or to even fail completely. Several efforts to overcome this issue have recently been reported (e.g., [1], [2], [3], [4], [5], [6], [7], [8], [9]), most often focusing on a specific task and application.

We investigate the discrepancies between the simulated and real robot states for joint positions, velocities and torques along the trajectories of free space movements, and propose *Realeasy*, a general, task-independent approach to model the residual between simulation and real environment.

*This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by Knut and Alice Wallenberg Foundation

[1]All authors are with the Dept. of Computer Science, Faculty of Engineering (LTH), Lund University, 221 00 Lund, Sweden alexander.durr,elin_anna.topp,volker.krueger@cs.lth.se

[2]https://sites.google.com/ulund.org/realeasy

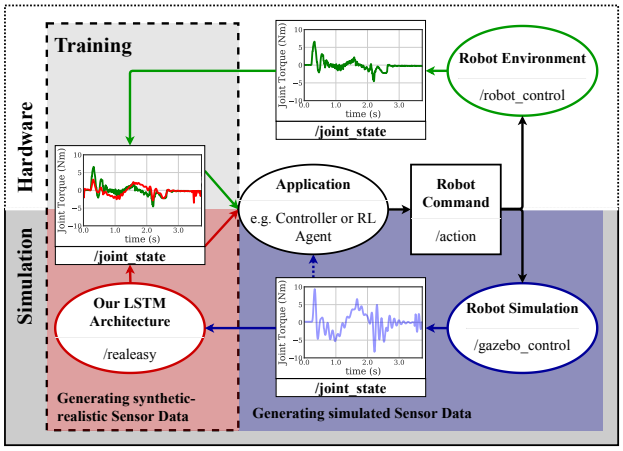

Fig. 1. Our *Realeasy* module integrated into the ROS framework (red), showing the learning pipeline to generate synthetic-realistic sensor data from comparison of real (green) and simulated (blue) robot data

Although discrepancies between real robot and simulated states might be easily visible for a human observer, formulating a filter or rule to handle them is non-trivial. Thus, we formulate a supervised learning problem based on two position-aligned sets of position, velocity and torque readings recorded from execution of the same trajectories once on a real robot and once in a simulated environment respectively. The dataset generation process for the supervised learning problem is automated and does not need human intervention. After having investigated several potential neural network architectures we propose a *systematically deduced explainable Long Short-Term Memory architecture* for translating the sequence of simulated joint states into a sequence of real joint states, as depicted in Fig. 1. We also provide a carefully designed and documented dataset for improving the learning rate and the model's performance. When deployed, the learnt model simply adds the predicted residual to the joint states along the simulated trajectory to generate synthetic realistic-looking sensor data (see Fig. 1).

Our comprehensive evaluations show a reduction in mean squared error in the residual of at least one order of magnitude. We also evaluate our trained model on a second real Panda robot to demonstrate the possibility of a direct transfer to other individual robots. Finally, we describe the benefits of our approach over possible combinations with state-of-the-art methods in the field of system identification [1], simulator augmentation [2], [3], [4], [5] and action grounding [6], [7], [8], [9].

In summary, our main contributions are:

1) two systematically developed neural network archi-

tectures to improve simulated robot state data, easily integrated into the ROS framework [10], see Fig. 1;
2) the design description for our data set for the Franka Emika Panda robot and the network parameters. In addition, we aim to share our data set, a real-time graphics-enabled docker image for robot control, reproduction of our results and direct reuse of our trained model.

II. Related Work

We present major insights of related work and present the differences to our approach with respect to the most popular simulation to reality transfer techniques.

A. System Identification

System identification (SI) is the area of identifying a set of dynamic parameters of a specific system, as presented for example by Gaz *et al.*, who solve the problem for the Franka Emika Panda robot through using penalty-based optimization [1]. With those parameters a robot physics simulator is improved. Their approach gives overall reliable results which are valid in most of the robot's work space, yet the limitations lie in the physics model itself which excludes dynamic effects like change in friction in the joints through movement over time. SI in general gives a good baseline for a simulator's parameters, but calibration for each individual robot can be required depending on the task. Furthermore, to ensure a good match the parameters of the simulation, e.g., masses of links, are the only free variables which can result in unrealistic results in favor of simulation and reality alignment. As a comparison, our approach can be based on realistic estimates of the simulation parameters, such as friction and link masses.

B. Simulator Augmentation

Tobin *et al.* [2] use Domain Randomization (DR) by imposing a prior distribution over the parameters of the simulator to find a robust control policy to achieve a certain task. DR has the disadvantage that it requires domain knowledge and handcrafting of the parameter randomization. Since the policy is robust for many possible sets of parameters, it finds a sub-optimal solution for reality and fails if the task at hand requires high precision. Our approach does not rely on parameter randomization as we align the simulation closely to reality. Yet, this general approach can be combined with ours, thus allowing smaller disturbances facilitating a near-optimal policy required for high precision tasks.

Zeng *et al.* [3] show with Tossingbot how a robot arm can learn to throw different objects precisely. Their work in the field of Domain Adaptation (DA) calculates the throwing velocity with a physics engine first, and then corrects the residual in velocity with a Convolutional Neural Network. Their work showcases a successful integration of simulated and real environment. Similar to our approach the simulator's state output is corrected. In contrast to their work, however, we capture several physical phenomena that require correction, such as friction and inertia based on a sequence of previous events. Our approach is general and does not need specification of which phenomenon needs correction.

Ajay *et al.* [4] implement a recurrent neural network (RNN) predicting a residual between simulated and real data in the case of planar pushing with focus on modelling uncertainty in prediction. The hybrid model consisting of a deterministic physics engine and a stochastic neural network generalizes to different objects and requires little data. They demonstrate that stochastic augmentation can improve the simulation to reality transfer of a learned policy.

Golemo *et al.* [5] show with their Neural Augmented Simulator (NAS) that an LSTM can learn with a given dataset describing the transitions (real state - action - simulated next state) as input to predict the next real state by observing the differences between simulated next states and real ones. When deployed, the next state outputted from the LSTM is used to hard-set the new initial state of the simulation, thus changing potentially non-smoothly positions, velocity and acceleration in the simulation. Obvious problems can be that by changing the position the robot might collide with objects in the simulated environment. We want to highlight their idea that the LSTM's cell state acts as the long-term memory of previous state-action-state transformations from simulation to reality. The NAS approach is closest to ours, yet the major difference is that our LSTM merely needs the simulated states as input, making the taken action implicit. We do not see a compelling reason why the action would be needed as explicit input for improving the Panda simulation. Furthermore, we do not need to hard-set the simulation to the LSTM's output since we align the trajectories in position. This ensures collision-free augmentation and speeds up the data collection significantly.

C. Action Transformation

Hanna *et al.* [6] show with their work in Action Transformation (AT) how sending transformed actions to the simulator can cause the simulated result to better match real execution. A forward model trained on real data, which gives for a state action pair the next state, is followed by an inverse model trained on simulated data, which gives for a (state, next state) pair the action required. They successfully demonstrate their framework with an RL agent trained for bipedal movement in the RoboCup league. Their Grounded Action Transformation (GAT) approach is based on the assumption that it is possible to alter the action to achieve the same state transition as desired by the policy in reality.

The same research group recently improved the GAT algorithm with Karnan *et al.* [7] Reinforced Grounded Action Transformation (RGAT), Desai *et al.* [8] Stochastic Grounded Action Transformation (SGAT) and Desai *et al.* [9] Generative Adversarial Reinforced Action Transformation (GARAT). Compared to GAT, RGAT trains by alternating between learning an action transfer policy while holding a target policy fixed and learning said task-specific target policy while keeping the action transfer policy fixed. SGAT introduces stochastic behavior for action transfer. GARAT is an adversarial approach where RGAT acts as the generator

and a discriminator learns the difference between RGAT output and real data.

These techniques work well given an initial guess for the target policy which solves the task, but the action transfer policy is only valid in the vicinity of the target policy's state and action space. In contrast, our approach is target policy agnostic and generalizes over the whole state and action space. Interestingly, our base robot simulator is already precise in position and velocity sensor readings. What is needed is an improvement of the insufficient performance in torque simulation without worsening position and velocity at the same time. In our case, changing the action means changing the torque applied to a joint of the Panda robot, but this would also change the position as well as the velocity.

The analysis of related work leaves us with only one possible angle of attack: the simulator's state output through DA for the following reasons:

- SI is limited by the model and its parameters which can not cover all present physical phenomena.
- DR results in sub-optimal policy making it unsuitable for high precision tasks.
- AT is unable to change position, velocity and acceleration independently.

III. Background and Notation

In this section, we describe the technical background for our approach and introduce some notation as referred to in the later parts of the paper.

A. The Robot System

The Franka Emika Panda robot has seven revolute joints with torque sensors, giving seven degrees of freedom (DOF). The torque sensors are on the side of the links, allowing direct measurement of the torque. Other robots approximate the torque by measuring voltage and current applied to the joint motors and have the issue of backlash in the gearbox between motor and link [11]. The robot can be controlled through the Franka Control Interface (FCI) by either sending position, velocity, or torque commands. For position and velocity commands an internal Joint Impedance or a Cartesian Impedance controller are available.

The dynamic model of a robot can be described by the Euler-Lagrange equation [12]

$$\mathbf{M}(\mathbf{p})\mathbf{a} + \mathbf{S}(\mathbf{p}, \mathbf{v})\mathbf{v} + \mathbf{g}(\mathbf{p}) = \tau \ . \tag{1}$$

with position $\mathbf{p}$, velocity $\mathbf{v}$, acceleration $\mathbf{a}$ and torque τ vectors of the size of the robot's DOFs, the inertia matrix $\mathbf{M}(\mathbf{p})$, the gravity vector $\mathbf{g}(\mathbf{p})$ and the Coriolis and centrifugal forces captured by $\mathbf{S}(\mathbf{p}, \mathbf{v})\mathbf{v}$. A physics simulator can solve the dynamic problem given the dynamic parameters for each link of the robot, consisting of link mass, center of mass, symmetric inertia matrix entries and friction. Retrieval of feasible dynamic parameters is a SI problem [1, for example], which we will not describe in detail here.

B. Long Short-Term Memory (LSTM) networks

To ground the description of our design for the *Realeasy* architecture, we will give a rather detailed explanation of recurrent neural networks and especially LSTMs [13]. A typical application of LSTM architectures is Natural Language Processing where they are either used for classification of text or to translate text sequences of one language into another. Hence, we want to clarify that this association with sequence to sequence learning as it is presented by Sutskever *et al.* [14] is not connected to our paper. A related field which uses LSTMs for regression is time series forecasting, i.e., the task of, given a history of temporal events $X_{<T}$, predicting the next N future events $X_{[T,T+N]}$.

In sequence to sequence regression learning, instead of predicting future values, an entire temporal sequence is mapped through regression to another temporal sequence. Based on a sequence X with elements $X_t, t \in \{0, \ldots, T\}$ an entire sequence Y over the same time steps is inferred.

For our approach as later described in detail, we assume the following: An LSTM cell at time t takes as input x_t, the output h_{t-1} and cell state c_{t-1} of an LSTM cell at $t-1$. The previous output h_{t-1} forms the short-term memory, whereas the previous cell state c_{t-1} forms the long-term memory which was passed and edited by all previous LSTM cells. The notation of an LSTM unit contains the input weights W_f, W_i, W_o, recurrent weights U_f, U_i, U_o, biases b_f, b_i, b_o and activation σ_f, σ_i, σ_o of the forget gate (2) [15], update gate (3), and output gate (4) respectively. A candidate cell state $\tilde{c}_t$ is calculated for the current inputs with the weights W_c, recurrent weights U_c, biases b_c and activation σ_c (5). To form the new cell state c_t (6), the forget gate vector f_t decides which entries of the previous cell state c_{t-1} to keep and remove. The update gate vector i_t decides on which entries of the candidate state $\tilde{c}_t$ to add to c_t. The output gate vector o_t selects the entries of the activation $\sigma_h(c_t)$ to form the output h_t (7). The forward pass of the LSTM is described as

$$f_t = \sigma_f(W_f x_t + U_f h_{t-1} + b_f) \tag{2}$$
$$i_t = \sigma_i(W_i x_t + U_i h_{t-1} + b_i) \tag{3}$$
$$o_t = \sigma_o(W_o x_t + U_o h_{t-1} + b_o) \tag{4}$$
$$\tilde{c}_t = \sigma_c(W_c x_t + U_c h_{t-1} + b_c) \tag{5}$$
$$c_t = f_t \circ c_{t-1} + i_t \circ \tilde{c}_t \tag{6}$$
$$h_t = o_t \circ \sigma_h(c_t) \ . \tag{7}$$

For a multi-layered LSTM architecture with $L \in \mathbb{N}$ layers the defined variables are indexed with the layer number l in the exponent. The first LSTM layer is denoted LSTM^0. One LSTM^0 cell has as input x_t^0 the element X_t of the time series X. For layers $l > 0$ the input is the output h_t^{l-1} of the LSTM cell in the layer below. The input weights W, recurrent weights U and biases b are shared between each LSTM cell of the same layer. The forward pass of one single LSTM cell of layer l is expressed as

$$h_t^l = \text{LSTM}^l(h_t^{l-1}, h_{t-1}^l) \text{ for } l > 0 \ . \tag{8}$$

The output of a full neural network architecture (NN) given a sequence X is simply denoted as $\text{NN}(X)$.

The specifics of our proposed LSTM architecture are explained in the following section, in particular in subsection IV-C.

IV. Approach and Method

Dynamic parameters identified through SI [1], give rise to the possibility to select a robust and stable robot controller to reach positions in the robot's work space. We can use the dynamic parameters as well in a robot physics simulator like Gazebo [6], [16], which returns satisfying position and velocity sensor readings as well as a tolerable estimate for torque in the robot's whole configuration space. Direct application of Machine Learning techniques to the robot's high-dimensional continuous state and action space would be a fruitless endeavor. We use thus DA to only learn the residual [3], [17] of the temporal position, velocity and torque sequence to correct the simulation output with a recurrent neural network architecture [4].

A. Custom Loss

As a measure of difference between simulated trajectory X and real trajectory Y we can apply the mean squared error (MSE) formula on the residual to describe a loss function. However, instead of minimizing the residual, our aim is to learn the residual as such to later add it to the simulated trajectory. Hence, we need a custom loss function, which we base on MSE and call mean squared residual error (MSrE)

$$\text{MSrE}(X) = \text{MSE}(X + \text{NN}(X)) \tag{9}$$

$$= \frac{1}{T}\sum_{i=1}^{T} \left(Y_t - (X_t + \text{NN}(X_t))\right)^2 \ . \tag{10}$$

Since we compare the elementwise difference of the two time series, other losses that deal well with outliers (mean absolute error or Huber loss) would be the wrong choice for our type of data.

B. Data set and Preprocessing

To apply our approach, we create a data set of simulated and real trajectory pairs with position, velocity and effort data for each of the robot joints. The trajectories are planned, executed and tested for feasibility on the real robot. If the trajectory plan is successful, it is also executed in simulation to create the pair. For training, the data set contains only joint movements to avoid trajectories with way points in this training data set. Since moves containing way points consist of short piece-wise joint moves, we can conclude that this brings no negative effect. Furthermore, to improve the learning rate, the data set contains a significant amount of single joint movements to increase observations of causal effects. If only one joint is moving, we observe no changes in position and velocity in all other joints, but a change in torque, which we are interested in. The remaining data set contains all possible combinations of joints moving together at the same time. All movements are at different speeds and fill the whole robot work space.

For testing, a test set sample from the original data set is drawn. To measure generalizing behavior, a second test set consisting of Cartesian linear moves containing way points is collected.

Preprocessing of the data set starts with alignment of the time series pairs by start point detection. This is possible when the recording contains the robot resting before and after trajectory plan execution by observing movements outside the confidence interval. Compared to time-series alignment with cross-correlation the first method is more precise and adds value. The robot accelerating and decelerating captures inertia and elastic behavior. Afterwards, the trajectories are cut and normalized by using the specifications of the robot joint limits, which makes the data set and model easy to improve with new data later on. Normalization of the data set to the interval $[0, 1]$ brings numerical stability to the chosen neural network architecture later on.

C. Neural Network Architecture Design Process

Starting simple, applying a fully connected dense neural network to a single element of the time series X can capture static differences in configuration, but without capability to have temporal information about previous movements it can not capture the dynamic effects we are facing.

RNNs function well for short-term time series, but lack in performance when considering causes of effects which show a long delay. Since typical robot recordings are at 100-1000Hz, this architecture can not be applied. LSTMs as introduced in III-B have the capacity to solve the task and our problem. For the activation of the forget, input and output gates σ_f, σ_i and σ_o, a standard sigmoidal function is chosen to keep the properties of forgetting and storing information by simple multiplication with a value in $[0, 1]$.

To be robust against outliers in the input data X, *Dropout* can be used to skip a percentage of the input. This also reduces the need for learning an informative and robust cell state. If x_t is skipped, it has an immediate effect on equations (2)-(5). This requires the previous cell state c_{t-1} to be informative enough to be used again for the current LSTM cell, promoting long-term memory. For the activation σ_c of (5) and σ_c of (7) the standard $\tanh$ function and the linear function were used, since they give a symmetric output in the positive and negative range. The maximally achievable absolute residual error is the distance between the joint limits. Given that we normalized the data set, this means that all possible values for the residual lie in $[-1, +1]$. With *Dropout*, the linear version becomes numerically unstable, whereas the stability is kept by the $\tanh$ function which only covers the feasible interval of the residual. This function behaves in the vicinity of its root linearly which is also a requirement to capture the linear-proportional character of the problem.

Considering a single LSTM layer LSTM^0, the first LSTM cell $\text{LSTM}^0(x_0, h^0_{-1})$ has an empty cell state c_0 at the beginning of a time series and will be filled with each new observation x_t. This results in inaccurate predictions at the beginning that improve along the time series until an

informative cell state is reached after which the performance plateaus. An additional layer on the same level, which runs from $t = T$ to $t = 0$, shows the opposite behavior of being inaccurate at $t = T$ and improving towards $t = 0$. Simple addition of the outputs does not resolve the issue, so we concatenate the outputs to pairs $[h_f^0, h_b^0]$ and add another bidirectional LSTM layer on top which takes the pairs as input, denoted $\text{LSTM}^1([h_f^0, h_b^0], h^1)$. If the forward LSTM layer LSTM^1 has an empty state, this layer disregards the first element of the pair h^0 since it is the output of the forward LSTM in LSTM^0. By summing the output of the second bidirectional Layer, we remove the start point issue and capture the temporal effects of the dynamic movement along the trajectory. When the robot is resting, we observe that this architecture predicts a constant offset from the real trajectory because the dynamic component is missing. To capture static differences between the trajectories, a final DENSE layer connecting the entries of output h_t^l is added.

To make it real-time capable and simplify the architecture, i.e., *Realeasy*, the bidirectional character is removed. If the problem allows a short period of initialization with no prediction, the cell state can be updated quickly to become informative within a few time steps. The LSTM architecture is only passing information forward in time. Thus, when a new state X_{T+1} is observed, the hidden states c_T^0, c_T^1, h_T^0 and h_T^1 of the previous LSTM cells LSTM^0 and LSTM^1 can directly be used to calculate the new prediction in real-time. Our experiments show that this computation consisting of two consecutive (8) calculations takes less than 0.39ms, since the matrix-vector multiplications in (2)-(5) are 21-dimensional.

D. Training

When the data set contains trajectories in which the robot is always resting at the beginning and end of the trajectories, the LSTM would learn after how many steps to rest. To remove this effect completely, the windowing technique is applied. Windowing takes a random sub interval of a trajectory and uses it for training. We use a mix of learning in random batches of trajectory intervals, where the shortest trajectory of a batch is determined. A fraction of the shortest length is used as the sub trajectory length. For other, longer trajectories in the batch a sub trajectory is drawn. This method results in a mix of longer and shorter trajectories. Since we built our LSTM architecture bottom up, focused on capturing the bare minimum of behavior, we did not give it the capacity to overfit on the training data. Batch normalization has no effect on the learning. Experiments with regularizers show that the L1 regularizer can not be used in any configuration.

E. Inference

Control based on the bidirectional version of *Realeasy* for precise calculation of the residual is shown in Algorithm 1. The bidirectional layers are applied to a generated time series from the simulator and results in an overall precise performance especially at both ends of the time series compared to a plain feed forward approach. This architecture can be useful for short time sequences, since the calculation effort which comes with the backwards LSTM is data intensive. The upper layers have to wait for the lower ones. The output for each new time step is a time series of residuals of the past states. The controller or agent can make decisions based on the whole time series of residuals.

Algorithm 1 Precise *Realeasy* LSTM Integration

procedure CONTROL ON PRECISE INFERENCE
  while execution = true **do**
    $\mathbf{h}_{t,f}^0 = \text{LSTM}^0(\mathbf{x}_t, \mathbf{h}_{t-1}^0, \mathbf{c}_{t-1}^0)$
    $\mathbf{h}_{t,b}^0 = \text{LSTM}^0(\mathbf{x}_t, \mathbf{h}_{t-1}^0, \mathbf{c}_{t-1}^0)$
    $\mathbf{h}_{t,f}^1 = \text{LSTM}^1([\mathbf{h}_{t,f}^0, \mathbf{h}_{t,b}^0], \mathbf{h}_{t-1}^1, \mathbf{c}_{t-1}^1)$
    $\mathbf{h}_{t,b}^1 = \text{LSTM}^1([\mathbf{h}_{t,f}^0, \mathbf{h}_{t,b}^0], \mathbf{h}_{t-1}^1, \mathbf{c}_{t-1}^1)$
    $\mathbf{h}_t^1 = \mathbf{h}_{t,f}^1 + \mathbf{h}_{t,b}^1$
    $\tilde{\mathbf{x}}_t = \text{DENSE}(\mathbf{h}_t^1)$ ▷ generate synth. state
    $\mathbf{u} = \text{CONTROLLER}(\tilde{\mathbf{x}}_\mathbf{t})$ ▷ use synth. state
    $\mathbf{x}_{t+1} = \text{EXECUTE\_ON\_ROBOT}(\mathbf{u})$
  end while
end procedure

Algorithm 2 Load and Initialize *Realeasy* LSTM

function INITIALIZE_HIDDEN_STATES( )
  Load *Realeasy* LSTM model
  Load *Realeasy* Parameters W, U, b
  Initialize h_{-1}^l and c_{-1}^l
  for $t \leq T_{\text{Warmup}}$ **do**
    $\mathbf{h}_t^0 = \text{LSTM}^0(\mathbf{x}_t, \mathbf{h}_{t-1}^0, \mathbf{c}_{t-1}^0)$
    $\mathbf{h}_t^1 = \text{LSTM}^1(\mathbf{h}_t^0, \mathbf{h}_{t-1}^1, \mathbf{c}_{t-1}^1)$
    $\mathbf{u} = \text{CONTROLLER}(\mathbf{x}_\mathbf{t})$
    $\mathbf{x}_{t+1} = \text{EXECUTE\_ON\_ROBOT}(\mathbf{u})$
  end for
  return $\mathbf{h}_{T_{\text{Warmup}}}^l \mathbf{c}_{T_{\text{Warmup}}}^l$
end function

For real-time calculation we remove the Bidirectional part of the LSTM architecture and remain with two forward pass LSTM layers LSTM^0 and LSTM^1. Algorithm 2 describes the initialization of the *Realeasy* model, where the controller decides directly on the simulator state x_t for a period of time T_{Warmup}, because of the empty cell state c_0 resulting in bad performance as discussed in Sec. IV-C. Algorithm 3 describes the integration into real-time control. After initialization of the hidden states, the forward LSTM is capable of giving satisfactory results. At each new time step the cell can reuse the hidden states of the previous step. The output at each new time step is one residual for the current time step.

V. EXPERIMENTAL SETUP AND RESULTS

We evaluate our approach extensively and quantitatively to show its value in improving the simulation of robot trajectory executions in free space. Regarding the direct applicability

Algorithm 3 Real-time *Realeasy* LSTM Integration

procedure REAL-TIME CONTROL ON INFERENCE
  $\mathbf{h}_0^l, \mathbf{c}_0^l$ =INITIALIZE_HIDDEN_STATES( )
  while execution = true **do**
    $\mathbf{h}_t^0 = \text{LSTM}^0(\mathbf{x}_t, \mathbf{h}_{t-1}^0, \mathbf{c}_{t-1}^0)$ ▷ Sec.IV-C
    $\mathbf{h}_t^1 = \text{LSTM}^1(\mathbf{h}_t^0, \mathbf{h}_{t-1}^1, \mathbf{c}_{t-1}^1)$
    $\tilde{\mathbf{x}}_t = \text{DENSE}(\mathbf{h}_t^1)$ ▷ generate synth. state
    $\mathbf{u} = \text{CONTROLLER}(\tilde{\mathbf{x}}_\mathbf{t})$ ▷ use synth. state
    $\mathbf{x}_{t+1} = \text{EXECUTE\_ON\_ROBOT}(\mathbf{u})$
  end while
end procedure

of our approach in a concrete learning scenario, we have restricted ourselves to theoretic considerations and comparisons with related methods, as we had periodically limited access to our laboratory during the covid-19 pandemic. These are presented in section VI. Here, we present the evaluation scenario, for which we made the design decision to develop the approach in a docker container [18], so that the solution is easy to share and our results are reproducible for others. To control the robot, all communications need to be in real-time, including those between the FCI, the connected computer, the docker container as well as the ROS nodes.

Our ROS framework for these experiments is configured so that the real robot and the simulated robot in Gazebo are controlled with joint trajectory controllers. For the real environment, the joint trajectory controller sends position commands to the FCI which uses the internal joint impedance controller to create joint torque commands to move the robot arm. We chose to use the FCI internal joint impedance controller to ensure the real robot trajectories to be reliably reproducible. For the simulated environment, the joint trajectory controller directly transforms the commands into torque commands. Internally a PID controller is used to approximate the robot behavior. Our approach will learn to correct for this difference on the lowest controller level, which is for both environments to operate at 1kHz. The sensor readings are recorded and processed on a higher level at 100Hz.

MoveIt Motion Planning [19] is used to generate trajectory plans and execute them with different velocity scaling. The collected data set contains random 850 multi-joint and 870 single joint movements which fill the robot's work space. Of this data set, 80% were used for training, 15% for validation, and 5% for testing. Additionally, another test set was collected, containing 50 different Cartesian linear movements in xz, xy, and yz direction.

We compared a manually fine-tuned set of parameters for Gazebo and controller Φ_0 with the parameters given by Gaz *et al.* [1] without damping (Φ_1) and with damping (Φ_2) to determine which dynamic parameters would bring the simulation closest to reality. A comparison of the MSE given the simulator parameters is presented in Table I. For the following experiments we use the manually fine-tuned set of dynamic parameters Φ_0 for the Gazebo simulator.

TABLE I

MEAN SQUARED ERROR OVER ALL ROBOT STATE VARIABLES FOR GAZEBO SIMULATOR OVER DYNAMIC PARAMETERS

Simulator Parameters	Φ_0	Φ_1	Φ_2
MSE	1.50e-3	1.549e-2	1.62e-2

Results

We show in Fig. 2 the torque sensor readings of an exemplary trajectory (in data set, No. 18) of the real robot (green), the simulated sensor readings (blue) and the improvement by our Precise *Realeasy* LSTM architecture (red). It is shown how the *Realeasy* model removes fluctuations from simulation while also adjusting to the behaviour of the real robot. Joint 1, 2 and 3 are moving, while the other joints remain still in position and velocity. The effects of one joint on the other joints are modeled precisely by our LSTM architecture. It also demonstrates how the model manages to capture characteristics particular to each joint, for example following the ground truth while being consistent for different magnitudes of torque.

The improvements of torque prediction are not unique to the presented example trajectory in Fig. 2, but improvement is seen across both test sets, as shown in Table II. The MSE of the synthetic-realistic torque readings for multi-joint moves is reduced by at least one order of magnitude. With the cartesian linear test set we show that our model can generalize to linear trajectories as well. A subset of the cartesian linear test set was evaluated on another Franka Emika Panda robot at Örebro University resulting in a reduction in MSE in torque of 53%.

For position and velocity data, where the simulation is already accurate, our model did not improve the accuracy. However, we argue that this is not a shortcoming of the model. Since residuals in position, velocity, and torque were all weighted equally in the loss function, the model practically only optimized for torque since this torque residual was several orders of magnitude higher. A user of our approach can choose to weigh position, velocity and torque differently.

VI. DISCUSSION

Let us summarize and connect the related work section II, approach and method IV and our experiments V.

SI is general and task independent but can result in unrealistic simulation parameters as seen in [1]. Our dataset as presented in Section IV-B is similar to the one used in their work in respect to that it can be collected before learning a control policy. Yet we allow any random moves, whereas typical SI approaches, including [1], rely on trajectories based on periodic motions as an assembly of different frequencies.The inflexibility of SI with the restrictive underlying kinematic model make it unable to capture the additional observed physical phenomena. If realistic but

TABLE II
MEAN SQUARED ERROR OF TORQUE SENSOR READINGS FOR EACH JOINT. SYNTHETIC-REALISTIC SENSOR DATA CREATED FROM OUR PRECISE *Realeasy* LSTM ARCHITECTURE.

Test Set	Joint		0	1	2	3	4	5	6
Multi-Joint	Simulation	$(Nm)^2$	3.64e+01	2.52e+01	2.42e+01	1.96e+01	1.23e+01	7.17e+00	7.15e+00
	Realeasy	$(Nm)^2$	**1.52e+00**	**2.84e+00**	**1.51e+00**	**1.56e+00**	**7.38e-02**	**1.02e-01**	**6.93e-02**
Cartesian Linear	Simulation	$(Nm)^2$	1.23e+01	6.67e+00	7.52e+00	5.34e+00	2.13e+00	2.60e+00	1.65e+00
	Realeasy	$(Nm)^2$	**9.02e-01**	**1.21e+00**	**9.21e-01**	**7.68e-01**	**5.89e-02**	**5.19e-02**	**1.18e-01**

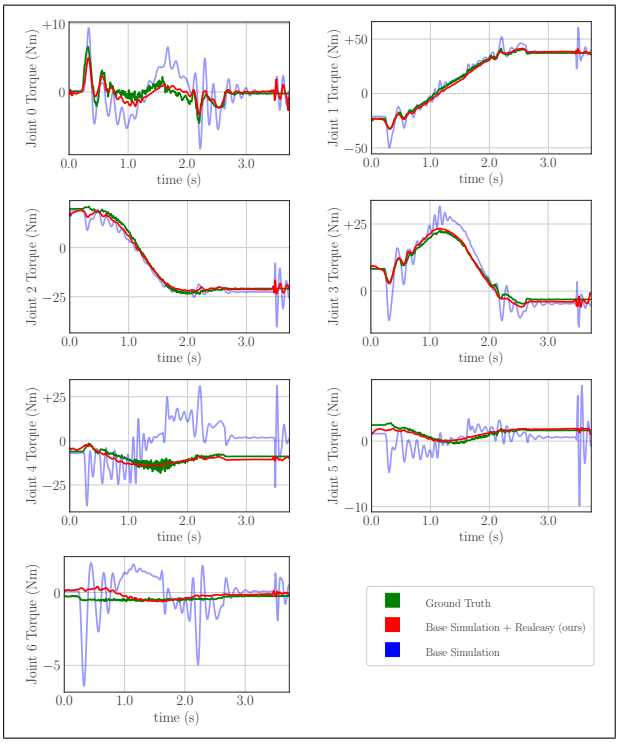

Fig. 2. The joint torques for the 7-degrees of freedom Franka Emika Panda robot in Nm. Ground truth shows real robot joint torque (green). The baseline simulation with gazebo shows vibrant behavior (blue), but follows the trend of the ground truth. The Precise *Realeasy* residual correction of the simulated sensor readings follows the ground truth closely.

guessed parameters are not good enough to fit the simulator model to reality, SI is necessary to find initial simulator parameters as the capabilities of our approach are restricted through its architecture (see Section IV-C). Our approach can be used on top of SI to diminish the inaccuracies that the basic model is unable to capture. This shows that SI and our approach can be seen as complementary.

The AT approaches [6], [7], [8], [9] are task specific, whereas ours is task independent as the augmentation is independent of the target policy. Furthermore, our approach does not only focus on desired simulator output but keeps position and velocity of the robot in the simulated environment intact, which is impossible for AT. We argue that by changing the action while leaving the simulator model as is, we can not create the observed realistic states. Changing parts of the state the simulator delivers as output are imperative, e.g. leaving position and velocity as is while modifying the torque. If one would just change the action, one of the variables will inevitably be unrealistic due to the unchanged underlying kinematic model.

Since the AT approaches are task specific and augment the simulation in the vicinity of the target policy, these approaches need less real training data than ours if an initial good guess for a target policy is present. If no such initial target policy working in reality exists, the AT approaches will fail. Our approach will capture the most important physical phenomena not covered by the kinematic model just from random movements generalizing in the whole state and action space. In contrast, AT captures the discrepancies between simulation and reality that are present when executing a target policy, leaving out general physical phenomena. This shows that one of our trained models can be shared between researchers to improve the simulator from the start, requiring no real training data at all. We suggest finding a target policy in the augmented simulator which is more likely to work well in reality and apply AT approaches to fine tune the target policy.

Inspired by [3], [4] and [5] we can confirm that in our case a hybrid model of physics simulator and residual neural network is a good choice to give valid results in the whole robot work space. Our custom loss (see Section IV-A) causes our approach to learn the residual that needs to be added to the simulator output. This restricts the range we generate as augmentation and assures that given a state input we will stay close to it's vicinity. Our approach differs from the previously mentioned work by a) applying DA to higher dimensional spaces and b) task independence. Compared to the closest approach to ours, NAS [5], we solely rely on state transitions without actions which makes our approach generalize well and we do not need to reset the simulator state after each iteration. We use the basic capabilities of an LSTM to capture physical phenomena and by systematically constructing an underfitting architecture as described in Section IV-C, we use it as a smart filter e.g. to diminish swinging and offsets as seen in Fig.2. A limitation of our approach is visible in the plots for joint 0 and 4 where the underfitting character dampens the swinging too much and takes some time to correct the offset of joint 6.

The approaches [5], [6], [7], [8], [9] cover a slightly different problem for transferring a policy from simulation to reality. Their augmentation is only valid for a given working initial target policy, or operate on a restricted state and action space, e.g., their state space has only position and

velocity but no torque and their actions are limited to position control. This allows AT and NAS to change actions as they disregard the effect on torque (AT) or allow discontinuity through resets of the simulator (NAS). For this reason a direct comparison of their approaches with ours in one benchmark is not possible or would lead to an unfair comparison.

Our experiments show a comparison between our approach and SI [1] as we want to showcase the improvements of the simulator quality in general that is valid over all states and policies applied.

VII. Conclusion

We considered the problem of subpar simulation, and propose and demonstrate an easy-to-use LSTM architecture for improving simulated data. The approach and method are formulated generally and are easy to adapt by other researchers to related problems.

We did not explore all possible neural network architectures to find an optimal architecture, but show that with considerations regarding the limitations of the robot, its sensors and physics, we could find a feasible solution. Our efforts result in an at first glance relatively small and uncomplicated neural network. After thorough examination we conclude that bigger would not have been better. The smaller network size naturally ensures a generalizing behavior of the network, while our results demonstrate that residual torque is consistently reduced in simulation. A well-generalizing solution is robust to changes of robot individuals and changes in types of motions. We leave our network and software open for other researchers to further investigate the specification of the loss function, the network itself, regularizers and their data set proportions. This work inspires to consider the simplistic approaches that can still be highly effective and efficient as demonstrated with our real-time capable implementation.

Potential extensions to this work are various because of its general character. With the mentioned related work in mind, investigation of Simulation to Reality policy transfer of control policies trained in RL can be a valuable extension that could make the connection with comparative benchmarks stronger. Yet we want to highlight that our approach is applied before policy training and merely makes the simulation more realistic through DA. Our approach is limited by the capability of the policy training algorithm to be able to deal with realistic data. We argue that removing unrealistic behavior such as swinging and offsets while improving simulated friction can only improve the transfer of a control policy trained in simulation. We also found indications that it is possible to use our trained model to identify situations when the robot physics simulator is far off from reality. This knowledge enables us to get a confidence measure for the simulation of robot manipulators. This can be used as guidance for learning approaches. We consider further investigations in this direction as future work.

Acknowledgment

We thank Todor Stoyanov for help with the robot setup and evaluation of our approach on another Panda individual. We thank Matthias C. Mayr, Mathias Haage, Björn Olofsson, Johannes A. Stork and Quantao Yang for helpful discussions.

References

[1] C. Gaz, M. Cognetti, A. Oliva, P. R. Giordano, and A. De Luca, "Dynamic identification of the franka emika panda robot with retrieval of feasible parameters using penalty-based optimization," *IEEE Robotics and Automation Letters*, vol. 4, no. 4, pp. 4147–4154, 2019.

[2] J. Tobin, R. Fong, A. Ray, J. Schneider, W. Zaremba, and P. Abbeel, "Domain randomization for transferring deep neural networks from simulation to the real world," in *2017 IEEE/RSJ international conference on intelligent robots and systems (IROS)*. IEEE, 2017, pp. 23–30.

[3] A. Zeng, S. Song, J. Lee, A. Rodriguez, and T. Funkhouser, "Tossingbot: Learning to throw arbitrary objects with residual physics," 2019.

[4] A. Ajay, J. Wu, N. Fazeli, M. Bauza, L. P. Kaelbling, J. B. Tenenbaum, and A. Rodriguez, "Augmenting physical simulators with stochastic neural networks: Case study of planar pushing and bouncing," in *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2018, pp. 3066–3073.

[5] F. Golemo, A. A. Taiga, A. Courville, and P.-Y. Oudeyer, "Sim-to-real transfer with neural-augmented robot simulation," in *Proceedings of The 2nd Conference on Robot Learning*, ser. Proceedings of Machine Learning Research, A. Billard, A. Dragan, J. Peters, and J. Morimoto, Eds., vol. 87. PMLR, 29–31 Oct 2018, pp. 817–828. [Online]. Available: http://proceedings.mlr.press/v87/golemo18a.html

[6] J. Hanna and P. Stone, "Grounded action transformation for robot learning in simulation," in *Proceedings of the 31st AAAI Conference on Artificial Intelligence (AAAI)*, February 2017.

[7] H. Karnan, S. Desai, J. P. Hanna, G. Warnell, and P. Stone, "Reinforced grounded action transformation for sim-to-real transfer," in *IEEE/RSJ International Conference on Intelligent Robots and Systems(IROS 2020)*, October 2020.

[8] S. Desai, H. Karnan, J. P. Hanna, G. Warnell, and P. Stone, "Stochastic grounded action transformation for robot learning in simulation," in *IEEE/RSJ International Conference on Intelligent Robots and Systems(IROS 2020)*, October 2020.

[9] S. Desai, I. Durugkar, H. Karnan, G. Warnell, J. Hanna, and P. Stone, "An imitation from observation approach to transfer learning with dynamics mismatch," in *Proceedings of the 34th International Conference on Neural Information Processing Systems (NeurIPS 2020)*, Virtual, December 2020. [Online]. Available: http://www.cs.utexas.edu/users/ai-lab?NEURIPS20-Karnan

[10] Stanford Artificial Intelligence Laboratory et al., "Robotic operating system." [Online]. Available: https://www.ros.org

[11] C. Gaz, F. Flacco, and A. De Luca, "Identifying the dynamic model used by the kuka lwr: A reverse engineering approach," in *2014 IEEE international conference on robotics and automation (ICRA)*. IEEE, 2014, pp. 1386–1392.

[12] W. Khalil and E. Dombre, *Modeling, identification and control of robots*. Butterworth-Heinemann, 2004.

[13] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[14] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Proc. NIPS*, Montreal, CA, 2014. [Online]. Available: http://arxiv.org/abs/1409.3215

[15] F. A. Gers, J. Schmidhuber, and F. Cummins, "Learning to forget: Continual prediction with lstm," 1999.

[16] N. Koenig and A. Howard, "Design and use paradigms for gazebo, an open-source multi-robot simulator," in *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)(IEEE Cat. No. 04CH37566)*, vol. 3. IEEE, 2004, pp. 2149–2154.

[17] A. Kloss, S. Schaal, and J. Bohg, "Combining learned and analytical models for predicting action effects," *arXiv*, 2018. [Online]. Available: https://arxiv.org/abs/1710.04102

[18] D. Merkel, "Docker: Lightweight linux containers for consistent development and deployment," *Linux J.*, vol. 2014, no. 239, Mar. 2014.

[19] S. Chitta, I. Sucan, and S. Cousins, "Moveit![ros topics]," *IEEE Robotics & Automation Magazine - IEEE ROBOT AUTOMAT*, vol. 19, pp. 18–19, 03 2012.