Deep Learning for Continuous Control

Bachelor Thesis

Assistant Professorship of Neuroscientific System Theory
TUM Department of Electrical and Computer Engineering
Technical University of Munich

University Professor
Prof. Dr Jörg Conradt
Assistant Professorship of Neuroscientific System Theory

Submitted by
Maximilian-Alexander Knaller
born on: 15th June 1994
Görresstraße 16
80798 Munich
+49 172 811 83 13

Supervisor
M.Sc. Michael Lutter

Final Submission
Munich, 24th May 2017
Abstract

One approach to trial intensive robotic learning, especially the even more data intensive Deep Learning, is the transfer of knowledge gained from virtual simulations to the real world. Therefore, researchers use sensory rich simulations such as the Neuro Robotics Platform developed within the Human Brain Project to simulate robot experiments. This bachelor’s thesis extends the Neuro Robotics Platform to rate-based neural networks. The extension is achieved through a TensorFlow - ROS interface, where TensorFlow is used to simulate rate-based neural networks. The interface and the TensorFlow models are capable of simulating the state-of-the-art Multi Actor-Learner system and processing data in real time. The complete system achieves control loop frequencies of greater than 500Hz for reasonable neural network sizes. Additionally, this work demonstrates the applicability of the system by implementing a robot steering controller and a reaching trajectory planner via Deep Learning.
## Contents

1 Introduction ....................................................................................................... 1

2 TensorFlow Integration .................................................................................... 3
   2.1 Related Work ............................................................................................. 3
      2.1.1 NRP Architecture ............................................................................. 3
      2.1.2 Deep Learning Systems .................................................................... 4
   2.2 Preliminaries on TensorFlow Integration ................................................... 5
   2.3 Programming Model and Basic Concepts .................................................... 6
      2.3.1 Computation Graph .......................................................................... 7
      2.3.2 Offline Training ................................................................................. 8
      2.3.3 Actor ............................................................................................... 8
      2.3.4 Learner ........................................................................................... 8
   2.4 Evaluation .................................................................................................. 10

3 Case Studies ..................................................................................................... 12
   3.1 Van der Pol Oscillator ................................................................................ 12
      3.1.1 Setup .............................................................................................. 13
      3.1.2 Results ........................................................................................... 15
   3.2 Reaching Spheres from Visual Inputs ........................................................ 19
      3.2.1 Setup .............................................................................................. 20
      3.2.2 Results ........................................................................................... 22
      3.2.3 Closed-Loop Evaluation .................................................................... 23

4 Conclusion ........................................................................................................ 24
1. Introduction

Robots successfully performed many tasks in industry and research. They achieved manufacturing automation[1], surgeries[2] and household chores[3], for example. However, all these tasks required extensive human labor and knowledge to engineer perception, planning and control algorithms. Even though engineers spent decades designing these systems, they lack generalization and adaptability to unknown situations. As a result, robots are still only used in highly constrained environments and often fail if exposed to new environments.

One approach to overcome generalization limitations of engineered systems is robotic learning. Robotic learning means to learn a task from experience rather than programming. The experience can be generated by imitation learning, where a human demonstrates the task or Reinforcement Learning (RL), where the robot performs trials and learns from rewards. Using traditional learning approaches like Policy Search[4] or Incremental Online Sparsification[5], robots learned compliant control performance as well as high tracking quality and were able to perform interactive tasks like table tennis[6]. However, these algorithms used low dimensional states as input and did not scale to high-dimensional inputs such as images.

Motivated by the recent breakthroughs in Deep Learning[7], which revolutionized Computer Vision (CV)[8, 9], Natural Language Processing (NLP)[10, 11, 12] and Reinforcement Learning (RL)[13], research showed that it is possible to learn tasks directly from visual input. Deep Reinforcement Learning (DRL) agents[14], asynchronously trained over twelve hours, were able to gain superhuman performance on Atari games and moreover reached nearly human performance on car simulators. Car simulators not only have more realistic graphics than simple arcade games, but also require the agent to learn the dynamics of the instance it is controlling. DL also showed impressive results in real world robotics, capable to scale to complex three-dimensional manipulation tasks, like placement[15], pouring[15] and grasping[16]. Door opening tasks[17, 18] were achieved without any prior demonstrations or manually designed representations. Only joint related features and goal restrictions were provided. To sum up, deep neural networks have shown transfer of knowledge through multiple tasks and are a thriving field in research.

Applying machine learning algorithms to robotic applications is non trivial as robotic systems have many constraints. Control is restricted to real time and trials are limited by the number of robots. Especially in academic research, the number of available robots is modest. Therefore, simulation of robotic learning becomes more important. Simulations have to be sensory rich so that learned parameters can be transferred to the real world. In the robotic domain progressive networks, trained in simulation, performed well in the real world and reduced sample complexity significantly[19].
To accommodate these new requirements, many are developing dynamic, scalable and sensory rich environment simulations, which provide methods to easily test, verify and evaluate ideas and approaches. Ideally, simulations allow Plug & Play (P&P), providing an easy transferability of the virtually learned model to the real world. While adequate tools exist to simulate either complex neural networks or robots and their environments, there is so far no tool that allows effective and efficient communication between brain and body models. As a result, building this bridge between two complex worlds needs a lot of effort and inconveniently consumes researcher’s time.

As a result, the Neuro Robotics Platform (NRP)[20], developed within the Human Brain Project (HBP)[21], aims to fill this gap. The NRP offers scientists and developers a software infrastructure allowing them to easily implement scalable and runtime efficient simulations, consisting of different tools connected by provided interfaces. Designed modular, the platform takes into account P&P requirements and easy expandability. Nevertheless, within the current implementation, brain models are limited to spiking Neural Networks using the Neural Simulation Technology (NEST) simulator [22], preventing the user from exploring the promising capabilities of rate based neurons.

This Bachelor Thesis should extend the NRP by adding TensorFlow[23] as simulator for rate-based neural networks. This integration was accomplished by the implementation of a scalable and user friendly Application Programming Interface (API) between TensorFlow and the Robot Operating System (ROS). The API is embedded in a real time capable and reusable ROS node, providing the user a maximum of functionality, while obtaining as much flexibility for specific TensorFlow implementations as possible. The thesis demonstrates the capabilities of the extended NRP through two case studies, presented in chapter 3 and evaluates the performance of the API in chapter 2.
2. TensorFlow Integration

This chapter gives an overview of systems used in literature by introducing the Neuro Robotics Platform (NRP) and current Deep Learning implementations. Afterwards, this thesis will draw conclusions and describe the implemented setup.

2.1. Related Work

2.1.1. NRP Architecture

The NRP[20] (schematic shown in Figure 1), connects existing tools for simulating brain models (orange), with detailed simulations of robot bodies and environments (green), to verify biological models and develop brain inspired control algorithms. Currently, the NRP uses Gazebo as a robot simulator and Neural Simulation Technology (NEST) as a brain simulator. Both simulators are connected via the Robot Operating System (ROS), which is industry standard and simplifies message passing. Additionally, ROS as intermediate communication layer enables Plug & Play (P&P) functionality. As a result, robot embedding in form of WeBots, Gazebo or real robots can be achieved if ROS interfaces exist.

![Figure 1: Schematic of the Neuro Robotics Platform architecture providing a range of interfaces for robotics (green) and brain simulation (orange), connected via Transfer Functions (sand) to translate domain specific representations.](image)

Gazebo simulations are represented by floating point numbers describing model poses, joint states, velocities and accelerations. NEST uses spikes to represent information. As both simulators are based on domain specific representations, one requires a translation between domains. As translation is experiment specific, it is performed via user defined Transfer Functions (TFc) that translate from numbers to spikes and spikes to numbers.
2.1.2. Deep Learning Systems

Two different architectures can be identified in the Deep Learning literature. The first architecture is commonly used for Computer Vision (CV) tasks[24], where the experience is available in large offline data sets. Typical CV networks fit easily on a single GPU memory, while the data sets do not. Therefore, the data is sliced and distributed over n learners to reduce data transfer times. Learners optimally are placed on separate GPUs and process sliced data independently and asynchronously to update network parameters. Even though learners work asynchronously and independently, gathered knowledge is shared by synchronizing the network parameters across GPUs via a parameter server. To summarize, parallelized learners speed up learning through distribution, e.g., Google trained the Inception-v3 model using 200 asynchronous learners and increased training throughput by factor 100[24].

![Figure 2 Schematic of common Deep RL architecture (Multi Actor-Learner system) consisting of M actor-world pairs independently gathering experience used by N asynchronous learners to update shared parameters on parameter server. This separation of experience generation and learning, decreases data correlation, cuts training time and improves real time capabilities. Experience, available in offline data sets, can additionally be feed over file systems.](image)

The second architecture (Figure 2) is common in Deep Reinforcement Learning (DRL), where the generated experience is dependent on the current world state and current actor parameters. Within this architecture, m actors (blue) interact with a copy of the task environment (green), taking decisions based on the current world state and shared network parameters. Generated experience is then buffered within the replay memory (experience). Parallel actors can be used to speed up experience generation and to decrease correlation of stored data, as each world state differs. Similar to the CV approach, n parallel learners (orange) draw random mini-batches from the replay memory to further decorrelate data and optimize shared network parameters. This parallelization, called Multi Actor-Learner[14], also separates experience generation from learning. Furthermore, parallelization prevents the blocking of actors and improves real time capabilities. Depending on the training method, 16 asynchronous actor-learners achieved training speedups between 12 and 24 times[14].
2.2. Preliminaries on TensorFlow Integration

Inspired by the architecture of the NRP and Deep Learning systems, this thesis aimed to connect both in a parallelized, modular, and user-friendly way. As shown in Figure 3, the connection is accomplished by a reusable ROS node (TFN). Therefore, the TensorFlow Node handles information passing between TensorFlow and ROS messages and manages the Deep Learning System. In comparison to presented stand-alone Multi Actor-Learner systems, actors do not interact with worlds directly. Instead, actors use the implemented TensorFlow-ROS API to connect with simulations or robots on the NRP. Even though communication in both directions (actions and world states) has to be serialized and transmitted via ROS messages, this approach allows the user to use world independent actor implementations as Transfer Functions are used for translation. The following section describes the high-level architecture of the TFN and associated sub implementations.

**Figure 3** Multi actor-learner system connected to NRP simulations via the TFN API. Actions and world states get serialized and transmitted via ROS messages, during transmission Transfer Functions are used to translate domain specific representations. Gathered experience is used by learners to update shared network parameters.
2.3. Programming Model and Basic Concepts

Figure 4 Flowchart of TFN main thread. Once the basic TensorFlow graph is build and offline training is completed, m actors and n learners are initialized in individual threads. Afterwards, the thread reacts to commands until shutdown.

Figure 4 shows an high-level work flow of the TensorFlow Node (TFN), providing functionality for the Multi Actor-Learner (MAL) system management. The TFN initially uses a task-specific and user implemented neural network model to build the TensorFlow computation graph. The building of the computational graph is followed by a potential offline learning stage. M actors and n learners are initialized in individual threads, allowing each instance to run asynchronous and independently. Afterwards, the thread reacts to commands until shutdown. Components are described in more detail in the following subsections.
2.3.1. Computation Graph

The computation graph is the base of every TensorFlow implementation and is build before runtime. A computation graph can be seen as an abstract description of computations, where each node of the graph represents a specific operation connected via vertices representing tensors. At runtime, the nodes, branches or the whole graph can be triggered to execute related operations. The computational graph of Multi Actor-Learner systems (Figure 5) consists of n identically learner and m identically actor branches. Therefore, TensorFlow’s in-graph-replication[25] allows the making of replicas of existing branches, requiring the user to build each branch type only once. The graph of Multi Actor-Learner systems is then built by replication of m actors and n learners, making the system expandable and easier to handle.

Figure 5: TensorFlow Computation Graph consisting of actors and learners, built using in-graph replication. Actors are triggered by reception of new data and return the related network output. Learners draw mini-batches out of the replay memory to update network parameters.

Actors are triggered by the reception of new data and gather experience by processing received world states and return the corresponding network output. Therefore, they have a very simple structure consisting only of the neural network model, which can be seen as the central node of the branch.
In comparison, learners need three additional nodes to actively adapt shared network parameters. These nodes represent the dequeue operation, the loss computation and the optimizer. Once triggered, a random mini-batch is drawn out of the experiment buffer by the dequeue node and fed to the neural network. The loss computation node then computes the error between actual and desired output of the network. Finally, the loss is used by the optimizer to update the shared network parameters by stochastic gradient descent.

By default, an Adam optimizer with standard parameters and a learning rate of $10^{-3}$ combined with an Mean Square Error (MSE) loss is used. Nevertheless, others can be chosen if needed.

2.3.2. Offline Training
Offline training (Figure 6.a) is executed on stored data sets before actors and learners are initialized and any interaction with either simulated or real worlds takes place. As stated in 2.1.2, such data sets do not fit on a single GPU memory. Therefore, data is sliced and distributed over n parallel learners to update shared network parameters using a stochastic gradient descent optimization. During offline training, the performance of the network is evaluated and results are logged. Training is executed until a stop condition arises. A loss threshold (to ensure a minimum of preceding knowledge for complex tasks), a maximum number of epochs or a time threshold are frequently used as stop conditions. Nevertheless, in an optimal case one would use early stopping to prevent overfitting. However, data sets are task specific and gathering a good one is labor intensive, as it often requires human labeling.

2.3.3. Actor
An actor (Figure 6.c) feeds the received world states to the related actor branch of the computation graph every time new data is available and publishes the corresponding neural network output via a ROS message. Additionally, the actor thread saves gathered experience to the replay memory. Executing actors is time critical, as the runtime determines the real time capability of the setup for robotic tasks. Therefore, this thesis will evaluate the framework with respect to computation time, in section 2.4.

2.3.4. Learner
Learners (Figure 6.b) work very similar to offline learning, as network parameters are updated using loss computation and optimization via stochastic gradient descent. However, learners do not use experience stored in offline data sets. Instead, mini-batches are randomly drawn out of the replay memory to independently and asynchronously update the shared network parameters on the parameter server. Synchronization between actors and the parameter server is accomplished using TensorFlow's shared variables[26].
Figure 6  
(a) TensorFlow Node (TFN) offline training thread used to train network models on offline data sets to ensure a minimum of preceding knowledge.  
(b) TFN learner thread using data randomly drawn from the replay memory, to independently and asynchronously update shared network parameters on the parameter server.  
(c) TFN actor thread publishing the network output of received world states via ROS messages and saving experience for online learning.
2.4. Evaluation

As the actor computation time is critical for the systems real time capability and control loop stability, this section evaluates the actor computation time with respect to different network sizes. The evaluations were performed on a NVIDIA TITAN GPU running one actor and one learner.

![Actor Computation Time](image)

**Figure 7** Evaluation of the TFN actor computation time with respect to width. As high-performance GPUs are able to process all neurons of a layer in parallel, performance stays nearly constant for the tested range, indicating a low influence of the width.

Figures 7 and 8, show the computation time for the actor loop (described in Figure 6.c), which contains receiving the data, computing the network output, publishing the ROS message and saving the sample to the replay memory. The interval denoted as others measures the time between the last command and the start of the next iteration. Hence, others is composed of sleep time as the minimum iteration time is set to one millisecond, delays due to the general interpreter lock or other system interrupts. As shown in both figures, the required time for receiving and saving the data is negligible. The most time consuming operations are the output computation of the neural network followed by the output publication. The average computation time is about one millisecond for all steps. However, there exists high variability (see error area in Figure 7, 8) caused by the interference computation. The maximum computation time is 5 milliseconds.

To analyze the scalability to different network sizes, the model dimensions were varied. The network width was varied from 25 to 500 neurons per layer and the number of hidden layers from 1 to 20. Therefore, the largest network has five million parameters, which is sufficiently large for networks in robotic applications. The end-to-end training only had 92000 trainable parameters[27].
When analyzing the computation time with respect to width, one can observe that the computation time is invariant to network width (Figure 7). This can be explained as network weights are stored on the GPU and additional neurons can be parallelized across Cuda cores. No additional data transfer or computation cycles are required. When analyzing the computation time with respect to the number of hidden layers (Figure 8), the computation time slightly increases. The initial increase in forward interference time is compensated by the reduction of sleep intervals until the computation time of others cannot be reduced. From there on, the computation time slightly increases. The increase can be accounted to the sequential nature of hidden layers that cannot be parallelized. However, the increase in computation time is sub-linear. Therefore, the actual computation time of the forward interference seems to be dominated by management overhead such as data transfer and thread dispatching.

All in all, one can summarize that the implemented network achieves control loop frequencies that are sufficient for most robotic applications.

![Figure 8 Evaluation of TFN actor computation time with respect to depth. Layer activation must be performed sequentially, resulting in an increase of computation time. Nevertheless, the increase is sub-linear for evaluated depths.](image_url)
3. Case Studies

To prove basic functionality of the TensorFlow Node and to showcase the capabilities of Deep Learning, this thesis presents two supervised learning tasks. The first task uses a simple Multilayer Perceptron (MLP) architecture to control a four wheeled robot. The second will describe and draw conclusions out of an attempt to control a robotic arm using joint states and visual input.

3.1. Van der Pol Oscillator

The main focus of the first case study was to validate proper functionality of the TensorFlow integration. Therefore, a comprehensible supervised learning task was chosen and simulated in an minimalistic Gazebo environment. Within this task, a four wheeled robot shall be navigated to a limit cycle modeled by a Van der Pol Oscillator (VPO), a non-conservative oscillator with nonlinear damping. A Multilayer Perceptron (MLP) is used to represent the mapping between current position of the robot and oscillator velocity. This mapping is given by the VPO and can be described by a system of differential equations

$$\begin{align*}
\dot{x} &= \mu(x - \frac{1}{3}x^3 - y) \\
\dot{y} &= \frac{1}{\mu}x
\end{align*}$$

(3.1)

where \( \mu \) stands for the oscillator damping.

The underlying vector field of Figure 9 shows the velocities of the VPO with a damping factor of one. Pictured sample trajectories were recorded from simulation using (3.1) combined with presented steering command conversions and serve as reference for latter evaluation.

Figure 9 Vector field representing corresponding oscillator velocities a damping \( \mu = 1 \). Four limit cycle trajectories, recorded from simulation using (3.1) and presented steering command conversion, are used for latter evaluation.
3.1.1. Setup
The Neural Network Architecture

The MLP, pictured in Figure 10 has a simple feedforward architecture, three hidden layers each 50 units with Rectified Linear Unit (ReLU) activation function and a linear output layer. The widely used ReLU activation function has a strong biological and mathematical motivation[28] and is described by equation (3.2).

\[
h_i = \text{max}(W_i^T h_{i-1} + b_i, 0) \tag{3.2}
\]

The activation of hidden layer \( h_i \) is determined as maximum value of the layer’s input and zero. The layer’s input is calculated using the activation of the previous layer \( h_{i-1} \) and parameters, weights \( W_i \) and bias \( b_i \) of layer \( i \). The network represents the oscillators velocities (3.1) and the current yaw. For more details on the TensorFlow implementation, please refer to the in-code documentation.

![Figure 10](image)

**Figure 10** Fully connected feed forward neural network (NN) with three hidden ReLU and a linear output layer, used for VPO and yaw representation

Steering Commands

The husky is controlled by two steering commands, linear velocity in x direction and angular velocity with respect to the z axis. However, these axes refer to the robot’s coordinate system requiring a conversion of the VPO velocities, as these are defined in respect to the global coordinate system.
Figure 11 Linear velocity of the husky is calculated by the projection of the VPO velocity $v_{VPO}$ (yellow) on the robot’s orientation vector $o_x$ (red).

Linear velocity is calculated by a scaled and clipped projection of the VPO velocity vector ($v_{VPO}$) on the orientation vector ($o_x$) of the robot, as shown in Figure 11 and described by equations (3.3 - 3.4).

$$v_{projection} = \frac{v_{VPO}^T o_x}{|o_x|}$$  \hspace{1cm} (3.3)

$$v_{linear} = \begin{cases} 
0.1 & v_{projection} \leq 10 \\
\frac{v_{projection}}{10} & 10 < v_{projection} < 40 \\
0.4 & v_{projection} \geq 40 
\end{cases}$$  \hspace{1cm} (3.4)

The angular velocity is described by a simple limited proportional controller (3.5), as for middle range angle errors one can simply use the scaled error itself and for larger values a constant showed the best results.

$$lim_{angular}(x) = \begin{cases} 
-2 & angle_{error} \leq 0.2 \\
10 * angle_{error} & -0.2 < angle_{error} < 0.2 \\
2 & angle_{error} \geq 0.2 
\end{cases}$$  \hspace{1cm} (3.5)
3.1.2. Results
This section describes the training setup and evaluates achieved performance.

Training
To allow a feasible comparison between offline and online training, both were trained using Mean Square Error (MSE) loss and stochastic gradient descent. Optimization was performed by an Adam optimizer with standard parameters and a learning rate of $10^{-3}$. Network parameters were updated on a batch size of five samples and the network performance was evaluated on a batch of 100 randomly drawn test samples. The test data set contained 2000 additional samples, not included in the offline data set of 5,000, within the state space of five times five meters, uniformly distributed samples. The Replay Memory contained a maximum of 500 samples, training was started with a minimum of 250 samples.

Evaluation
To verify the functionality and capability of the neural network it was first trained offline. Figure 12 shows the Mean Square Error (MSE) decay of an randomly initialized network. The MSE decays to approximately $10^{-1}$ with a feasible standard deviation and an a relatively smooth mean.

Figure 12 Decay of the offline training MSE to approximately $10^{-1}$. One iteration refers to a single parameter update using stochastic gradient descent. Mean and deviation is calculated over 6 trails.
Online learning (Figure 13) was performed using mini-batches randomly drawn out of the replay memory, resulting in an MSE of approximately $10^{-1}$.

![Figure 13](image)

**Figure 13** Decay of the online training MSE to approximately $10^{-1}$. One iteration refers to a single parameter update using stochastic gradient descent. Mean and deviation is calculated over 6 trails.

Figure 14 shows a comparison between offline (orange) and online (blue) learning. Online learning showed less standard deviation, which is unexpected as in most cases correlation of the online gathered data results in bigger standard deviations. While both training methods decay to approximately $10^{-1}$, evaluation of the local performance will point out differences and give a more detailed insight.

![Figure 14](image)

**Figure 14** Comparison of MSE decay between offline and online training. Both methods decay to approximately $10^{-1}$. One iteration refers to a single parameter update using stochastic gradient descent. Mean and deviation is calculated over 6 trails.
Figure 15 shows an absolute and relative error representation over the whole state space for online and offline training. While offline training results in high relative deviations in the center, the overall absolute error is nearly constant at 0.3 matching the training MSE. Online training produces lower relative errors, nevertheless absolute errors are nearly three times higher at the edges of the state space, indicating a fundamental challenge of online learning. The state space is not explored uniformly as the actor takes decisions based on the current network parameters. In this case study the actor learns to navigate to the limit cycle close to the center. Therefore, the actor gathers less experience in the outer area of the state space, resulting in position dependent performance fluctuations.

![Figure 15](image)

**Figure 15** Absolute and relative deviation of network VPO velocity representation from truth value. Offline training results in high relative deviations in the center, the overall absolute error is nearly constant at 0.3 matching the training MSE. Online training produces lower relative errors, nevertheless absolute errors are nearly three times higher at the edges of the state space, indicating a insufficient state space exploration in these areas.
Finally, comparison of the optimal trajectory (green) and the trajectory (orange) using the online trained neural network shows satisfying results, as each path ends in the limit cycle. Nevertheless, as pictured in Figure 16 trajectories differ in some areas, indicating the lasting Mean Square Error (MSE).

Figure 16 Comparison between optimal limit cycle trajectories and trajectories performed using the MLP.
3.2. Reaching Spheres from Visual Inputs

In our second showcase we want to extend the neural network architecture to control an Universal Robots (UR) 10 arm, pictured in Figure 17. The end effector of the UR10 shall reach spheres using joint states and visual input of the scene. The spheres are placed in a cuboid (0.5m x 1m x 0.5m) which is aligned centrally in front of the robot. Traditional robotics would solve such a problem by using a perception system to extract the sphere position. Based on this position, a trajectory planner plans a smooth trajectory to the sphere in joint space. This trajectory is then executed by an joint controller. In this showcase the neural network should replace the trajectory planner and the perception system.

![Figure 17](image)

**Figure 17** UR10 robotic arm in task environment, reaching a sphere using joint states and visual input

During an initial first open loop control problem (Figure 18.a) the convolutional network learns to extract sphere position from image data. The output of the convolutional network is fed to the fully-connected layers, mapping the position to joint space via the learned Inverse Kinematics (IK). In summary, in this task the network shall be trained to represent the sphere position in joint space.

![Figure 18](image)

**Figure 18** a.) Open loop control problem, the neural network learns to extract the sphere position from image data. b.) Closed loop control problem where the network only outputs the next step in a trajectory leading to the goal. Therefore, this closed-loop system should be able to react to moving spheres.
The second step is a closed-loop control problem (Figure 18.b) where the network only outputs the next step in a trajectory leading to the goal. Therefore, this closed-loop system should be able to react to moving spheres.

3.2.1. Setup

Joint Trajectory Controller

A joint trajectory controller attempts to interpolate the movement between trajectory waypoints. Interpolation is performed using linear, cubic or quintic splines depending on the description level of the single waypoints. Therefore, the description determines the continuity level of the trajectory movement. UR provides a ready-to-use ROS Joint Trajectory Controller[29], which was used to execute trajectories.

The Neural Network Architecture

Inspired by recent research[27], the neural network (Figure 19) contains three convolutional layers, followed by an flattened layer, which additionally concatenates the robots joint states. Two fully connected hidden layers are used to produce controller commands.

Figure 19 The network contains three convolutional layers, followed by an flattened layer that packs the multidimensional image into a one dimensional tensor. The tensor is concatenated with the joint states, then passed through two fully connected hidden layers to produce controller commands.
**Data Set**

To train a neural network in a supervised fashion one needs a supervising signal. For this simple reaching problem MoveIt!\(^{[30]}\) was used as supervising signal. MoveIt! performs motion planning, to execute a desired continuous movement. In other words, MoveIt! breaks down the reaching task into a set of discrete waypoints, known as a trajectory. Waypoints at least consist of positions and the specific time instants they shall be reached. Optionally, waypoints can include velocities and accelerations.

Additionally, training needs a sufficiently large and usable data set, containing network inputs and corresponding supervising signals. However, gathering such a data set is challenging and labor intensive. Trajectories must be divided into their single waypoints and executed stepwise, as for each step the corresponding visual input is needed. Slicing trajectories is costly though, because trajectories are planned to be executed at once and include time related description. Therefore, one can not only plan a single trajectory and hand over waypoints separately.

To overcome these challenges this thesis uses the fact that planning is independent from execution and delimits the gathering of planning data and corresponding visual input. A trajectory is planned and evaluated algorithmic by extracting roughly time equidistant waypoints. Evaluation is necessary, as trajectories often contain waypoints that are very close in joint space and transitions between those do not hold significant information. Afterwards, the UR10 arm is moved to each way point of the evaluated trajectory by planning and executing sub trajectories, where the corresponding visual input is saved.

Using this approach a data set of 2300 trajectories was recorded automated, resulting in 14,000 samples consisting of the visual input, goal position, current joint states and the corresponding quintic trajectory waypoint.
3.2.2. Results

Training

Training was executed offline using Mean Square Error (MSE) loss and stochastic gradient descent. Optimization was performed by an Adam optimizer with standard parameters and a learning rate of $10^{-3}$. Learning and evaluation used an batch size of ten randomly drawn samples. The data set described in section (3.2.1) was randomly sliced into training (70%) and test set (30%).

Open-Loop Evaluation

Figure 20 shows the decay of the MSE of the neural network representing the goal position in joint space using only visual input. The MSE decays fast and smoothly to approximately $10^{-2}$. Indicating good representation of the goal position.

Figure 20 Decay offline training MSE trained on goal position extraction in joint space using only visual input.

To receive a meaningful error representation, the network was additionally evaluated in Cartesian space. Figure 21 shows the mean position error in centimeter for three different z values. Results are satisfying as errors are sufficiently small within the state space. Nevertheless, higher errors at boarders of the state space indicate inadequate covering by the data set.
Besides the numerical evaluations the behavior was verified in Gazebo, showing that the UR10 was able to reach the spheres in the open loop setup.

### 3.2.3. Closed-Loop Evaluation

While the offline MSE decayed to approximately $10^{-3}$, simulation of the task showed insufficient performance. The endeffector initially moves towards the goal, but gets stuck close to the center of the state space. This indicates lack of generalization, as performance did not match the MSE decay. This lack can most likely be traced back to the training data. Even though the data set contains 2300 trajectories leading to uniformly distributed goal states, these were all recorded from the same initial pose of the robot. Therefore, most trajectory way points are settled in the center of the state space, resulting in an insufficient covering of the state space. As a result, a new data set has to be recorded and evaluated, ensuring a uniformly distribution of waypoints rather than goal states. As this would be beyond the scope of this thesis, development was stopped at this point.
4. Conclusion

In this thesis, the integration of rate-based neural networks into the NRP was described, implemented, evaluated and applied. The implemented system combines the web-interface and the concept of transfer functions from the NRP with the Multi Actor-Learner system, which is used in state-of-the-art Deep Learning (DL) systems [18, 31, 17]. The implemented system is real time capable because the actor control loops achieve control frequencies greater than 500 Hz for reasonable network sizes. The computation time is invariant to the network width as these additional dimensions can be parallelized on the GPU. Only the number of hidden layers yields a sub-linear increase of computation time. The number of hidden layers cannot be compensated by parallelization as these are sequential operations. However, this negligible sub-linear increase shows that the overall computation time is dominated by overhead such as thread dispatching or data transfer.

After the system implementation, the system functionality was demonstrated by two case studies. The first case study controlled a four wheeled Husky robot to a limit cycle. The robot was able to learn the steering commands from online exploration and a supervised learning signal. The second case study let a robotic arm reach a sphere using visual input. For this task a data set was created and a network architecture implemented. For the open-loop experiment, where the network maps visual input to the sphere position in joint space, the learning was successful. The network converged to a solution and the robot was able to reach the spheres in simulation. For the closed-loop experiments the training converged, but the robot did not demonstrate the expected behaviour and got stuck at some joint positions. This unwanted behaviour was caused by a bad training set.

In the future it will be necessary to evaluate the system performance on multi-gpu systems, which could be used to distribute learners and actors to further accelerate actor control-loops and learning. Additionally, it will be necessary would need to record a better training set for the robotic reaching task. In the long term, one should evaluate the state-of-the-art Deep Reinforcement Learning algorithms[16, 17, 31] and Transfer Learning[19, 32] approaches within this framework.
List of Figures

Figure 1  Schematic of the Neuro Robotics Platform architecture ................................ 3
Figure 2  Schematic of common Deep RL architecture (Multi Actor-Learner system) ..... 4
Figure 3  Multi actor-learner system connected to NRP.............................................. 5
Figure 4  Flowchart of TFN main thread ................................................................... 6
Figure 5  TensorFlow Computation Graph ................................................................. 7
Figure 6  Sub implementations of TFN..................................................................... 9
Figure 7  Evaluation of TFN actor computation time with respect to width ............ 10
Figure 8  Evaluation of TFN actor computation time with respect to depth ............ 11
Figure 9  Optimal limit cycle trajectories with corresponding vector field ............... 12
Figure 10 Fully connected feed forward neural network (NN) for Van der Pol Oscillator representation .......................................................................................... 13
Figure 11 Velocity computation for Van der Pol Oscillator ........................................ 14
Figure 12 Offline training MSE for Van der Pol Oscillator ........................................ 15
Figure 13 Online training MSE for Van der Pol Oscillator ......................................... 16
Figure 14 Comparison of MSE decay between offline and online training for Van der Pol Oscillator ...................................................................................................... 16
Figure 15 Absolute and relative deviation of network VPO velocity representation .... 17
Figure 16 Comparison between optimal limit cycle trajectories and trajectories performed using the MLP ............................................................................................. 18
Figure 17 UR 10 task environment ............................................................................ 19
Figure 18 Control loops for Reaching Spheres from Visual Inputs ............................. 19
Figure 19 CNN for showcase: Reaching Spheres from Visual Inputs ......................... 20
Figure 20 Decay offline training MSE for Reaching Spheres from Visual Inputs ....... 22
Figure 21 Mean position error in centimeter for Reaching Spheres from Visual Inputs.... 23
# List of Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
<td>2, 5</td>
</tr>
<tr>
<td>CNN</td>
<td>Convolutional Neural Network</td>
<td>25</td>
</tr>
<tr>
<td>CV</td>
<td>Computer Vision</td>
<td>1, 4</td>
</tr>
<tr>
<td>DL</td>
<td>Deep Learning</td>
<td>i, 1, 12, 24</td>
</tr>
<tr>
<td>DRL</td>
<td>Deep Reinforcement Learning</td>
<td>1, 4, 24</td>
</tr>
<tr>
<td>GPU</td>
<td>Graphics Processing Unit</td>
<td>4, 8, 10, 11, 24</td>
</tr>
<tr>
<td>HBP</td>
<td>Human Brain Project</td>
<td>i, 2</td>
</tr>
<tr>
<td>IK</td>
<td>Inverse Kinematics</td>
<td>19</td>
</tr>
<tr>
<td>MAL</td>
<td>Multi Actor-Learner</td>
<td>i, 4–7, 24, 25</td>
</tr>
<tr>
<td>MLP</td>
<td>Multilayer Perceptron</td>
<td>12, 13, 18, 25</td>
</tr>
<tr>
<td>MSE</td>
<td>Mean Square Error</td>
<td>8, 15–18, 22, 23, 25</td>
</tr>
<tr>
<td>NEST</td>
<td>Neural Simulation Technology</td>
<td>2, 3</td>
</tr>
<tr>
<td>NLP</td>
<td>Natural Language Processing</td>
<td>1</td>
</tr>
<tr>
<td>NN</td>
<td>Neural network</td>
<td>1, 2, 7, 8, 13, 15, 19, 20, 25</td>
</tr>
<tr>
<td>NRP</td>
<td>Neuro Robotics Platform</td>
<td>i, ii, 2, 3, 5, 24, 25</td>
</tr>
<tr>
<td>P&amp;P</td>
<td>Plug &amp; Play</td>
<td>2, 3</td>
</tr>
<tr>
<td>ReLU</td>
<td>Rectified Linear Unit</td>
<td>13</td>
</tr>
<tr>
<td>RL</td>
<td>Reinforcement Learning</td>
<td>1, 4, 25</td>
</tr>
<tr>
<td>ROS</td>
<td>Robot Operating System</td>
<td>i, 2, 3, 5, 8–10, 20</td>
</tr>
<tr>
<td>TFc</td>
<td>Transfer Functions</td>
<td>3, 5</td>
</tr>
<tr>
<td>TFN</td>
<td>TensorFlow Node</td>
<td>5, 6, 9–12, 25</td>
</tr>
</tbody>
</table>
U
UR - Universal Robots ................................................................. 19–21, 23

V
VPO - Van der Pol Oscillator ...................................................... 12–14, 17, 25
Bibliography


