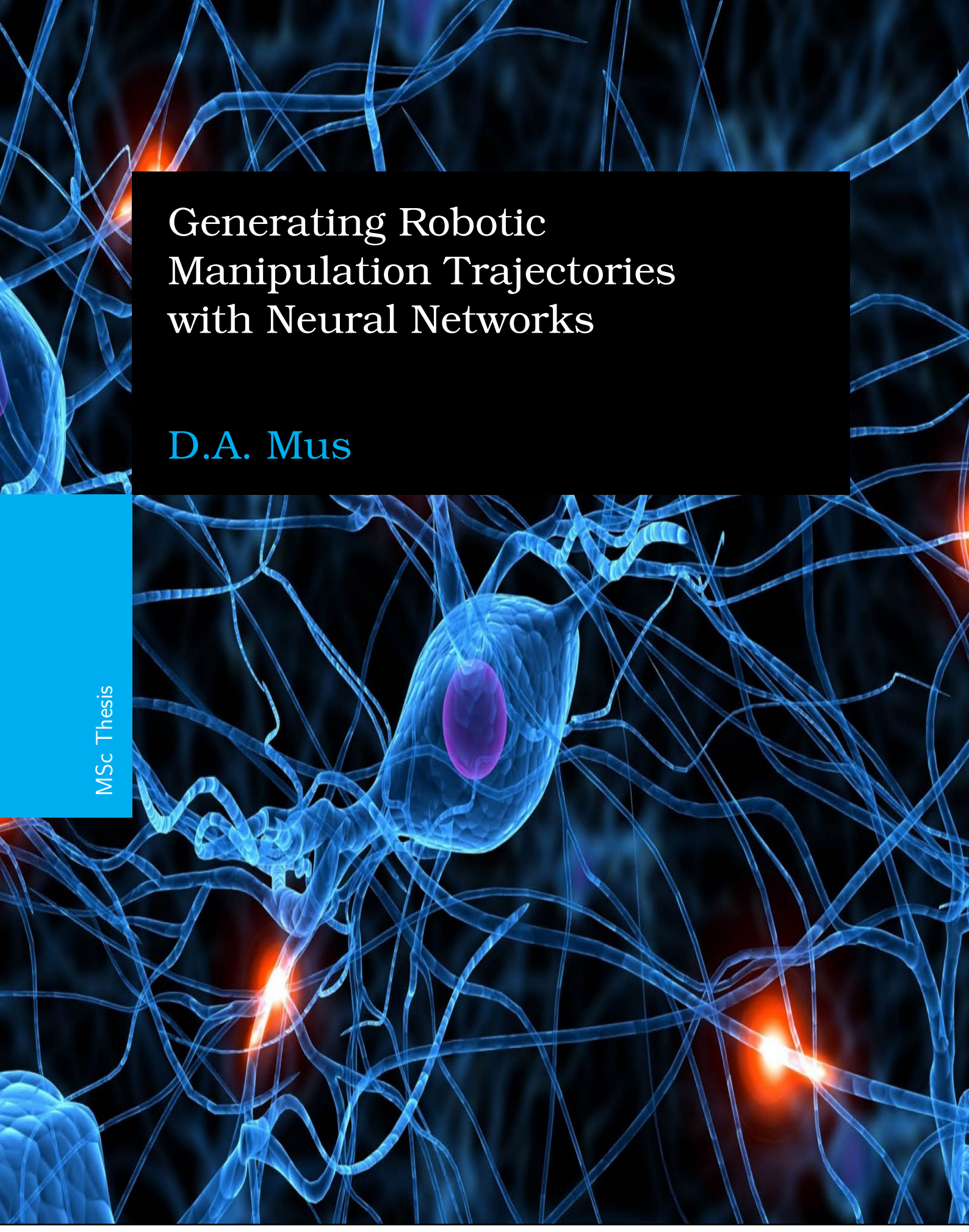


Generating Robotic Manipulation Trajectories with Neural Networks

D.A. Mus

MSc Thesis

TU Delft Delft University of Technology

Delft Center for Systems and Control

Generating Robotic Manipulation Trajectories with Neural Networks

MSc Thesis

D.A. Mus

March 22, 2017

Faculty of Mechanical, Maritime and Materials Engineering (3mE) · Delft University of Technology

The work in this thesis was supported by IBM. Their cooperation is hereby gratefully acknowledged.

Abstract

In order to interact with environments and appliances made for humans, robots should be able to manipulate a large variety of objects and appliances in human environments. When having experience with manipulating a certain object or appliance, a robot should be able to generalize this behaviour to novel, but similar objects and appliances.

When a human has to do a simple task like pushing a previously unseen button or rotating a knob, it has an idea of how to do this. Based on prior experience with similar object parts, a concept for this kind of manipulation tasks is formed.

In this work we use a similar idea to learn robots to infer how an object or appliance should be manipulated. We make use of a neural network approach to generate manipulation trajectories for a robot. An instruction in natural language and a pointcloud of the 'manipulatable' object part are encoded into a compact feature representation. We use a recurrent neural network to generate a manipulation trajectory, conditioned on this learned feature representation.

We report on experimental results with our model, first letting our recurrent neural network hallucinate manipulation trajectories. This shows that it has learned reasonable patterns. Then we compare the generated trajectories, conditioned on the learned feature representation, with the current state of the art. We show that for some simple tasks our model generates better trajectories, but in general does not have enough training data to generate reasonable trajectories for more challenging and complex tasks.

Table of Contents

List of Figures

List of Tables

Acknowledgements

I would like to thank my supervisor Jens Kober from the TU Delft for his assistance during the experiments and writing of this thesis. Furthermore I would like to thank my supervisor from IBM Zoltan Szlavik. I am also grateful to Robert-Jan Sips (IBM) and to Guy Lev and Einat Kermany (IBM Haifa) for their suggestions. In addition I would like to thank Jaeyong Sung from Stanford University for providing his code and helpful comments.

Delft, University of Technology D.A. Mus
March 22, 2017

Chapter 1

Introduction

A robot operating in a real-world environment should be able to do common tasks, like opening a door, turning a light on (Figure 1-1) or preparing a cup of coffee. In this way it could help or replace humans and make our life easier. We want to generate robotic manipulation trajectories for these kind of manipulation tasks. The robot has to learn how to perform these tasks, because we do not want to program the robot again for each new task. The robot has to learn this in such a way, that it is able to generalize its learned behaviour to new unseen environments. So the robot should be able to deal with novel objects and appliances, which it has never seen before.

Robots capable of doing tasks, in environments made for humans, could have a great impact on society. Elderly people and people with disabilities could be helped in their own household environments, which is very useful in times when the average age of people is increasing. Another example is production that is moved to low-wage countries that can be moved back, because now robots could perform these, for humans simple and repetitive, tasks. And consider the time people have for other activities, when robots are doing the 'boring' household tasks.

1-1 Learning Strategies

When we talk about learning, according to Atkeson [4], we can roughly identify four different learning strategies:

- *Learning by being taught.* A teacher (or programmer) tells the robot explicitly what to do in various situations by teaching or programming rules.
- *Learning by imitation.* The robot learns by observing (human) experts doing the task or related tasks.
- *Learning by thinking or dreaming.* The robot learns the task by planning or mentally practicing the task. While doing the task, the most appropriate plan is chosen and modified to the current situation.

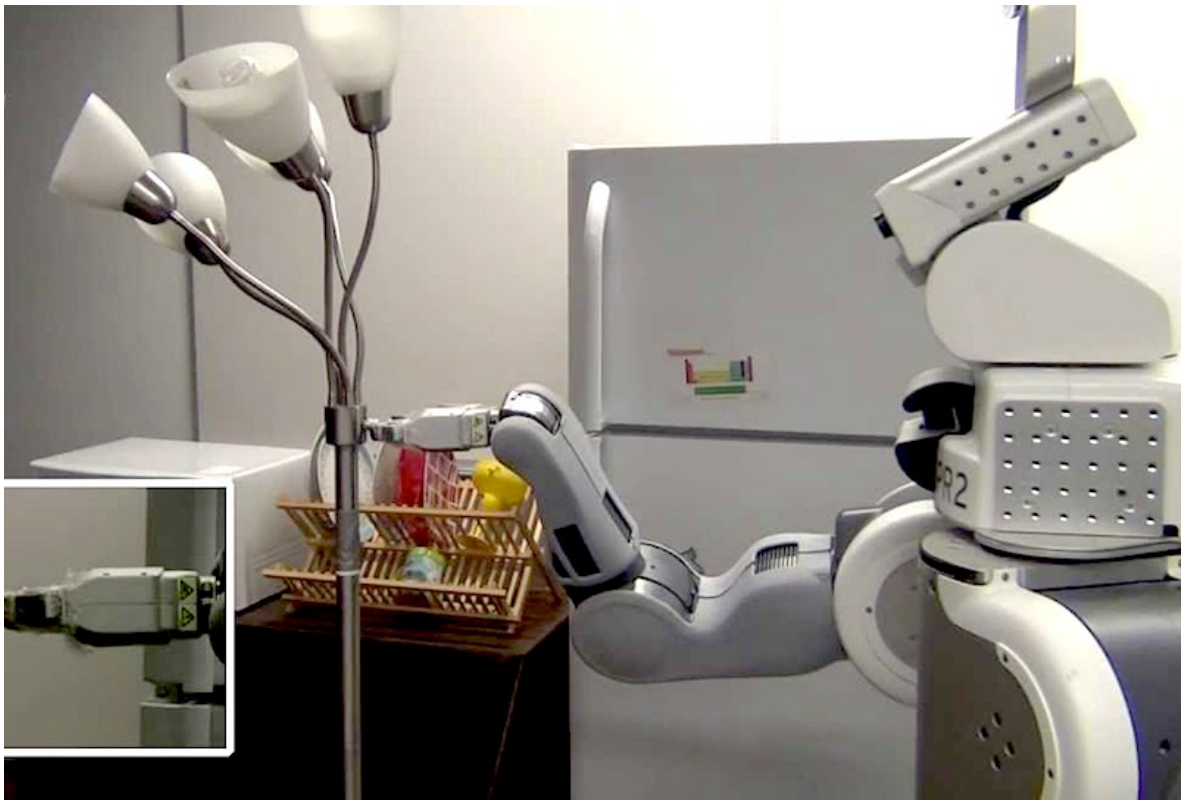


Figure 1-1: The PR2 robot turning a light on by rotating a knob. Image from https://majalah.ottencoffee.co.id/robobarista-robot-yang-jago-membuat-kopi/.

- *Learning by doing.* The robot learns from practicing the task.

In this work we propose a method which can be considered as a learning by imitation strategy (also called *learning from demonstrations* or *imitation learning* [5]). It needs a lot of demonstrations during its training phase to develop an understanding of how to generate trajectories, in such a way that it is able to generalize the observed behaviour to new unseen objects and appliances. We do not focus on the other strategies. Explicitly programming rules will never achieve enough generalization capabilities for novel unseen objects and appliances. Learning by thinking typically involves a model or simulation in which a step or action is evaluated and we consider the case where this is not available. Learning by doing can be a very powerful method, but requires a lot of trials to achieve good performance, when starting with near-zero knowledge [6]. However, this could be a good option to improve the behaviour learned by our strategy, when using that as a starting point.

In our learning strategy we want a model that is able to generate a manipulation trajectory. A manipulation trajectory can be considered as a sequence of waypoints. At each step the model thinks about or plans the most appropriate next waypoint. The plan for how to choose the next waypoint is learned internally in the model, during a training phase. For this training a let of demonstrations are required. The model is trained to build a trajectory step by step, by iteratively predicting the next waypoint.

We choose for this one waypoint per step approach, because we want to make use of the

sequential structure in trajectories. Our model generates trajectories step-by-step, with each next step based on the previously generated steps. Because we have no feedback involved in our method, it could be considered as an open-loop controller. However, this step-by-step method is particularly useful for eventually extending the method to use feedback, which can be provided after each step.

An alternative would be to generate a complete trajectory with a fixed number of waypoints in one go. In this way no use is made of the sequential structure, the network has to find out itself that waypoints would be related in a temporal way in the representation. This yields an unnecessary complex learning task, with a less compact model. Also it is less simple to include feedback, because then the whole trajectory has to be replanned.

1-2 A Neural Network for Generating Sequences

We choose a deep learning model to accomplish the generation task. Deep learning models have shown to be able to learn high level abstrations from data [7]. Therefore they are an appropriate candidate to learn high level features from training trajectories and use this to generate new robotic manipulation trajectories.

A special kind of a deep learning model, that is specialized in handling sequences, is a recurrent neural network (RNN). RNNs are a powerful class of dynamic models. They have been used to generate complex sequences with long-range structure in domains as diverse as text [8, 9, 10, 11], handwriting synthesis [2, 12], audio [13, 14, 15, 16] and image caption generation [17, 18]. All sequences with complex and long-range structures. We are interested in generating another kind of sequences, also with a complex structure, manipulation trajectories for robots.

In this work we investigate how to apply RNNs for the generation of robotic manipulation trajectories for a robotic arm. We look at an RNN 'hallucinating' about trajectories, based on all trajectories it has seen during training. Here there is no instruction or environment involved, the RNN is completely free in what it wants to generate. We extend this to conditioned generation. In this extension the network is allowed to predict its sequence prediction on a learned representation of the task and environment. This results in an architecture with two main components: (i) a model that learns a representation for a given task and environment and (ii) an RNN that generates a trajectory. In our work the task is represented in natural language and the environment as a pointcloud. We focus on component (ii) and use an existing implementation [3] for component (i). This existing implementation encodes high-dimensional inputs (natural language instructions and pointclouds) to a compact vector representing the task and environment. Then this compact representation — which should include the necessary information to generate a trajectory for this particular task — is feeded to the RNN. The RNN can then generate a trajectory, based on this representation of the task. So if we look to the complete model, we have an encoder which maps high-dimensional inputs (an instruction in natural language and the environment as a pointcloud) to a compact representation. The RNN (the decoder) maps the compact representation subsequently to a manipulation trajectory. So our complete model (an encoder-decoder architecture) generates a manipulation trajectory given a natural language instruction and a pointcloud of the environment.

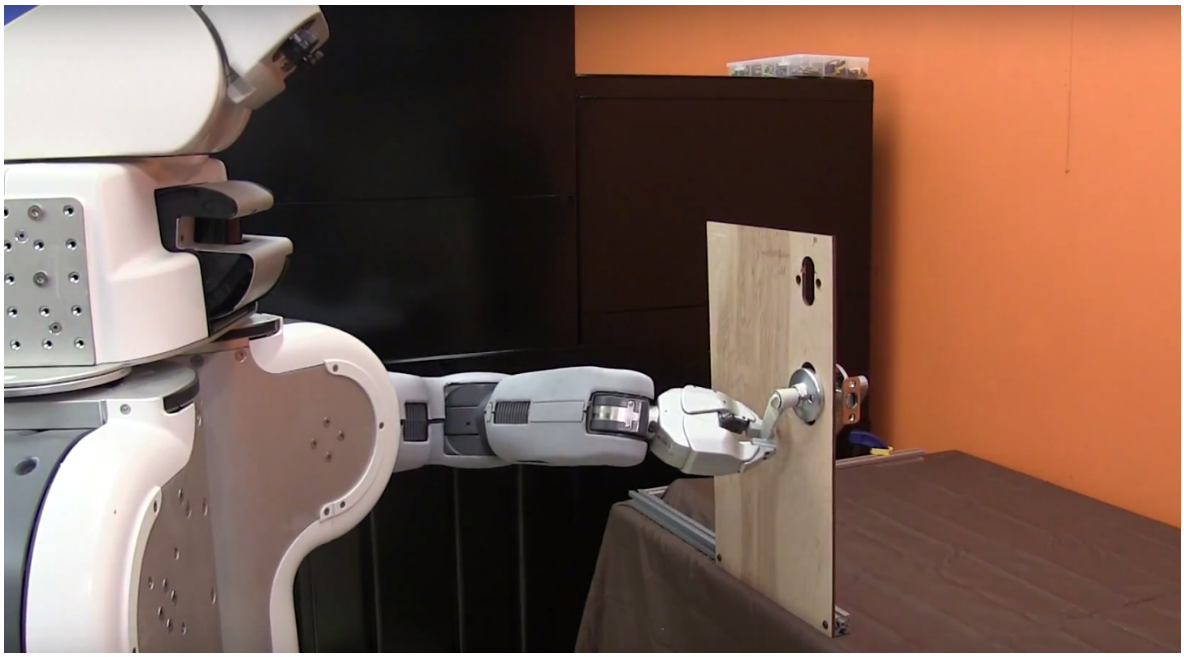


Figure 1-2: The PR2 robot opening a door. Image from https://sites.google.com/site/icra17dme/.

RNNs can be used for sequence generation by training them on a dataset of real data sequences. One step at a time is processed by the network. The network is trained to predict what step comes next. New sequences can be generated from a trained network by iteratively sampling from the output distribution of the network. Here we assume the predictions are probabilistic and form a probability distribution for the next step of the sequence. The sample from the output is feeded as input at the next step. Intuitively, the sampled predictions are treated as if they were real, much like 'dreaming' or 'hallucinating'.

For the trajectory generation, we consider the case without any interaction with the environment. There is no feedback. This means the trajectory generation is feedforward and purely offline and can be thought off as dreaming about a trajectory to fulfill a task, but without really executing it. When there is interaction with the environment possible, a form of reinforcement learning (learning by doing) can be used to practise the task and fine-tune performance. However, think of manipulating a new object or appliance in a household environment, for example opening a door handle (Figure 1-2). Based on experience with other handles a human can think almost unconsciously of a manipulation trajectory for this new door handle, much like 'dreaming'. This makes it interesting to apply a sequence learning method with recurrent neural networks to this kind of manipulation trajectories.

1-3 Challenges

The difficulty with current learning algorithms is that they can train an agent to perform very well in one particular task. The challenge is to have an agent than can transfer or generalize knowledge to different tasks. When it has learned to manipulate a number of

objects and appliances it should be able transfer this knowledge to manipulate novel objects and appliances. When we compare it to human learning, humans do not start from scratch with every new task. We are able to build on what we already know. We can apply the knowledge acquired across a whole series of experiences to each new task.

We train a recurrent neural network on a dataset with a large variety of small manipulation trajectories in a common household environment [19]. This includes trajectories to rotate a knob, pull a handle, press a button or hold a cup below a nozzle. We want our network to learn features about these trajectories, such that it is able to generalize this understanding to generate new trajectories. In this case the network is completely free in what it generates. When adding a representation (or context vector) of the task and environment the generation of a trajectory is not completely free anymore and we can speak of conditioned generation, which is an even more challenging task.

1-4 Related Work

By letting the trajectory generation be conditioned on a learned representation of the task and environment (the context), we see a correspondence to the encoder-decoder framework [18]. The environment can be a high-dimensional input, like a 3D point-cloud of the object that has to be manipulated. The task can be a instruction in natural language, like 'Rotate the knob clockwise to the desired setting'. A compact feature representation can then be inferred from these two inputs. That compact feature representation, or context vector, encodes the task and environment in a compact way. The encoder-decoder framework has a lot of successful applications, including machine translation [9, 10, 11], image caption generation [17], video clip description [18] and speech recognition [15, 16]. In an encoder-decoder structure both the input and output have a rich structure and are related somehow. All these encoder-decoder networks are based on a shared set of building blocks. Whereas the first systems had some form of multimedia content (images, video, audio) as input and text as output, new research shows even more promising directions with complicated rich structures as images [20] as output. This makes it a promising idea to try to use trajectory generation as another building block in encoder-decoder networks.

We apply our network to the Robobarista dataset [19]. This dataset contains crowd-sourced manipulation trajectories for a lot of different small tasks in a household environment. The dataset contains 1225 manipulation trajectories, which are demonstrated for 250 tasks and environments. In [3], a representation for the task and environment is learned. Besides this, a representation for each trajectory is learned that maps to the same feature space. During inference, when presented with a previously unseen representation for a task and environment, a suitable trajectory has to be selected. From the set of existing trajectories, the one closest in the feature space is selected. This does result in a reasonable performance, but by using already existing trajectories with a nearest neighbor approach, the flexibility and creativity seems a bit limited. These existing trajectories are originally executed in a probably similar, but different environment. So the selected trajectories are not adjusted to the small differences between the original and the new unseen environment. Furthermore the trajectories are used as they are, so there is no way to use a different starting position. By using recurrent neural networks we show that we can generate more flexible and creative trajectories.

1-5 Contributions

The main contributions of this work are:

- We extend the work on sequence learning to a new domain, robotic manipulation trajectories.
- We propose a new method for offline trajectory generation. Without any interaction with the environment, a trajectory is generated for a given task and environment. This method is applied to the Robobarista dataset and compared to the current state of the art.

1-6 Outline

In Chapter 2 required background material we use in later chapters is presented. In Chapter 3 we describe our method to generate trajectories with recurrent neural networks. In Chapter 4 we describe our method to build an encoder. Experiments and results with our RNN model are presented in Chapter 5. We summarize and discuss the methods and results in Chapter 6. Furthermore we give directions for possible future research.

Chapter 2

Background

In this chapter we describe the building blocks from deep learning we build upon later in this work. With deep learning we mean a powerful set of techniques for learning in neural networks. With neural networks we mean a biologically-inspired programming paradigm which enables a computer to learn from observational data. According to [7], deep learning allows computational models that are composed of multiple processing layers to learn representations of data with multiple levels of abstraction. Learned representations often result in much better performance than can be obtained with hand-designed representations [21]. For a more complete introduction to deep learning and neural networks we refer to [7] and [21]. This chapter focusses on deep learning methods we use for our method. This includes recurrent neural networks (Section 2-1), mixture density networks (Section 2-2) and encoder-decoder networks (Section 2-3). Furthermore we describe the work of Graves on handwriting [2] (Section 2-4), which combines the methods mentioned and is therefore closely related to our work. The current state-of-the-art approach for selecting a manipulation trajectory for the Robobarista dataset is described in Section 2-5. In Section 2-6 we compare the different deep learning methods for generating (sequential) data.

A neural network is typically a function that maps an input x to an output y. It is typically organized in layers, see Figure 2-1. Layers are made up of a number of interconnected 'units', which contain an 'activation function'. Data is presented to the network via the 'input layer', which is linked to one or more 'hidden layers'. The linking between units is done via weighted 'connections'. These weights are modified during training by a learning rule called 'backpropagation' [21]. The hidden layers link to an 'output layer' where the output is presented.

2-1 Recurrent Neural Networks

A special form of a neural network, specialized in handling sequential data, is a recurrent neural network (RNN). In an RNN, unlike feedforward neural networks, connections between units form a directed cycle. This creates an internal state of the network between inputs.

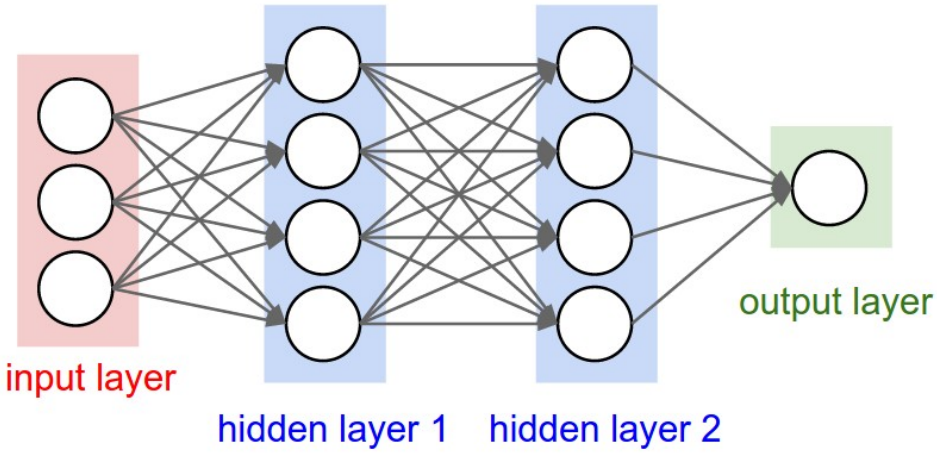


Figure 2-1: A basic feedforward neural network with an input layer, two hidden layers and an output layer. The layers are linked with weighted connections. Each unit (except for the input layer) contains an 'activation function'. Image from [1].

RNNs are called recurrent, because they perform the same task for every element of the sequence. The output depends on the previous computations. Another way to think about RNNs is that they have a "memory" (the internal state) which captures information about what has been calculated so far [22]. Recurrent Neural Networks (RNNs) have become the first choice for generating sequential data [2]. Recurrent neural networks achieve impressive results in language modeling [8], machine translation [9, 10, 11], speech recognition [15, 16], handwriting generation [2, 12], image caption generation [17, 18], audio generation [13, 14], etc.

The complete RNN computes a function from input histories $x_{1:t}$ up to step t to output vector y_t. The function is parameterised by weights W. An example architecture for an RNN is shown in Figure 2-2. As input we have a sequence of vectors $\mathbf{x} = (x_1, \ldots, x_T)$, for T steps. The input is passed through a stack of N recurrently connected hidden layers to compute the hidden vector sequences $\mathbf{h}^n = (h_1^n, \ldots, h_T^n)$, the internal state or "memory", with n indicating the hidden layer. Then the output vector sequence $\mathbf{y} = (y_1, \ldots, y_T)$ is computed for all T steps.

The hidden layer activations are computed by the following equation for $t = 1$ going to T and $n = 2$ to N:

$$h_t^1 = \mathcal{H}(W_{ih^n} x_t + W_{h^n h^n} h_{t-1}^n + b_h^n) \tag{2-1}$$

where the W terms are the weight matrices, W_{ih^n} connects the inputs to the nth hidden layer, $W_{h^n h^n}$ connects nth hidden layer to the nth hidden layer at time $t-1$, the b terms denote bias vectors and $\mathcal{H}$ is the hidden layer function.

Given the hidden sequences, the output sequence can be computed by

$$\hat{y}_t = b_y + \sum_{n=1}^{N} W_{h^n y} h_t^n \quad , \tag{2-2}$$

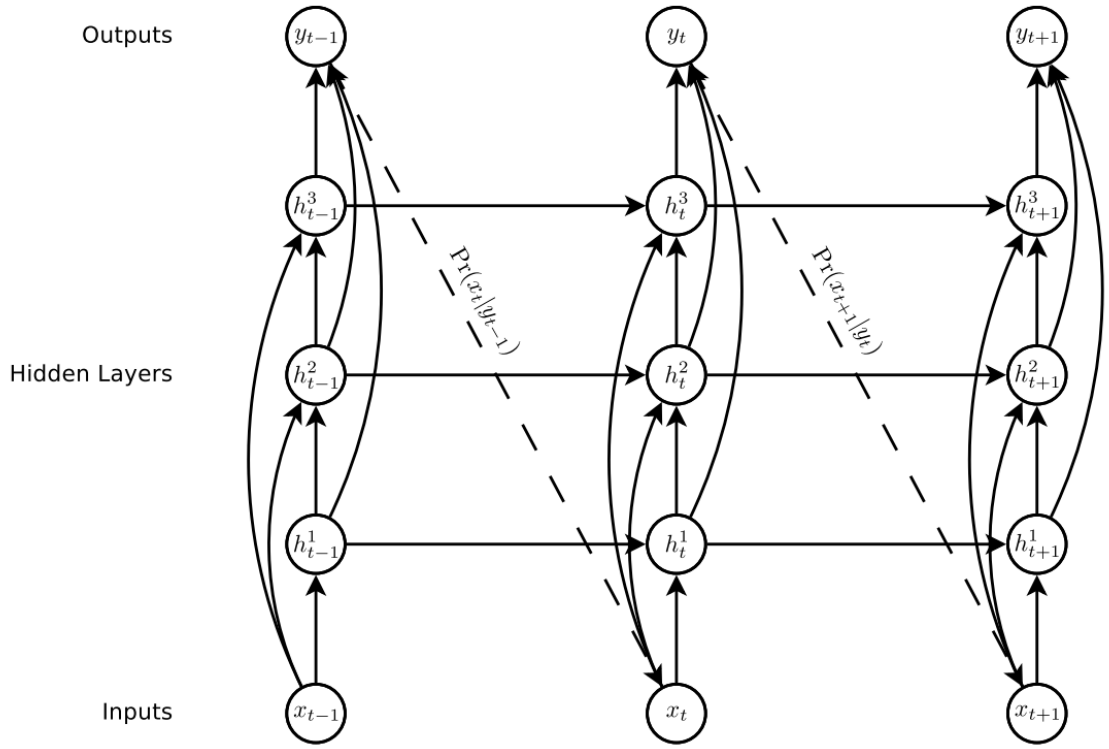


Figure 2-2: An architecture for a recurrent neural network with 3 hidden layers. Note here also the skip connections from inputs to all hidden layers and from all hidden layers to the outputs. This is a possible extension for an RNN, that makes it easier for information to flow directly to one of the top layers. Image from [2].

$$y_t = \mathcal{Y}(\hat{y}_t) \tag{2-3}$$

where $W_{h^n y}$ connects the last hidden layer to the output layer and $\mathcal{Y}$ is the output layer function.

The output vectors y_t are used to parameterise the predictive distribution for the next input, $\Pr(x_{t+1}|y_t)$. It can be very challenging to find a good predictive distribution for high-dimensional, real-valued data, more on this in the next section.

With the predictive distributions for the next inputs $y_{1:t}$, the probability given by the network to the sequence $\mathbf{x}$ is

$$\Pr(\mathbf{x}) = \prod_{t=1}^{T} \Pr(x_{t+1}|y_t) \quad . \tag{2-4}$$

To train the RNN a certain loss function has to be minimized. The probability of the sequence can be maximized via the maximum likelihood principle. This can be done via the negative logarithm of $\Pr(\mathbf{x})$, this is the function we want to minimize:

$$\mathcal{L}(\mathbf{x}) = -\sum_{t=1}^{T} \log \Pr(x_{t+1}|y_t) \quad . \tag{2-5}$$

The partial derivatives of the network weights with respect to the loss function can be computed with backpropagation through time [23]. Then the network weights can be trained with gradient descent.

So the network is trained to predict a predictive distribution y_t for the next input x_{t+1}. This can be used to generate a novel sequence. By iteratively sampling from the predictive distribution y_t and feeding this as the next input x_{t+1}, novel sequences can be generated.

In principle an RNN with enough capacity should be sufficient to generate sequences of arbitrary complexity. In practice, it turns out RNNs are unable to store information about past inputs for very long [2]. This limits its capabilities to model long-range structures. The problem is that when the predictions of the network are only based on the last few inputs, and these inputs where themselves also predicted by the network, the network has no ability to recover from mistakes in the past.

Having a better and longer memory can solve this issue, because it has a stabilizing effect. With a better memory it can look further back in the past to formulate predictions. Long Short-term Memory (LSTM) [24] is an architecture designed to be better at storing and accessing information than standard RNNs. By using LSTM memory cells in the hidden layers instead of "normal" units, storing and accessing information becomes easier for the network. Then the network becomes better at finding and exploiting long range dependencies in the data.

A problem with recurrent neural networks is that of exploding or vanishing gradients [2]. Because of the backpropagation through time a gradient is multiplied by the same weight matrix over and over again. To avoid exploding gradients, a commonly used method is to clip gradients such that they lie within a predefinied range.

2-2 Mixture Density Networks

We can use an RNN, possibly with LSTM memory cells, to generate a sequence. For each step t an output vector y_t is produced. This output is a predictive distribution for the next step of the sequence x_{t+1}. We do not want to directly output this next step, because then always the same sequence would be generated. The stochasticity involved by picking samples yields a distribution over sequences. We want a distribution over sequences that models uncertainty in a prediction and also gives different choices for each step.

In a *mixture density network* [25] the outputs are used to parameterise a mixture distribution. Each mixture component has a mean and covariance matrix. A subset of the outputs of the network is used to define weights for each mixture component. The remaining outputs correspond to the means, standard deviations and correlations for the individual mixture components. The number of mixture components is another hyperparameter of the model and, intuitively, is the number of choices the network can make for the next output.

So, when a mixture density network is used, the output of the network at step t is

$$y_t = (\pi_t^j, \mu_t^j, \sigma_t^j, \rho_{t\,j}^j)_1^M \tag{2-6}$$

for M mixture components. Here μ indicate means, σ standard deviations and ρ correlations.

When mixture density networks are used in combination with recurrent neural networks the output distribution is conditioned on the current input *and* the history of previous inputs.

2-3 Encoder-Decoder Models

We described how to generate sequences with a recurrent neural network and how to use a mixture density network to parameterise a predictive distribution for each step of the sequence. Generating sequences in this way means hallucinating about sequences, but there is no way to guide the sequence generation. Suppose for example the generation of manipulation trajectories *conditioned* on the manipulatable object and corresponding natural language instruction. We want a trajectory that is suited to perform the task described in the instruction for the corresponding object. Therefore the RNN, that generates the trajectory, needs additional information. It needs to know the position of the object, what kind of object, what kind of instruction, etcetera. We need a way to capture the relevant information from the raw and high-dimensional inputs. Here encoder-decoder models are very useful. The encoders maps the high-dimensional inputs to a compact representation, that is used by the decoder (the RNN that generates the trajectory).

Suppose a rich high-dimensional input structure and an output sequence that is related. Examples are machine translation, speech recognition and image captioning. Encoder-decoder networks were originally proposed in the context of machine translation, where a natural language sequence (the input) has to be mapped to a natural language sequence in a different language (the output) [9, 10]. In an encoder-decoder structure the encoder f_{enc} maps the input data $\mathbf{x}$ into a fixed-length vector representation in a continuous space. We call this the context vector $\mathbf{c}$:

$$\mathbf{c} = f_{enc}(\mathbf{x}) \quad . \tag{2-7}$$

The choice of f_{enc} depends on the type of input. A convolutional neural network (CNN) may be a natural choice for an image. When $\mathbf{x}$ is a natural language sentence a recurrent neural network (RNN) may be used.

Then the decoder f_{dec} generates the output $\mathbf{y}$ conditioned on the vector representation, or context $\mathbf{c}$. We could also say the conditional probability distribution of y given x is computed:

$$\Pr(\mathbf{y}|\mathbf{x}) = f_{dec}(\mathbf{c}) \quad . \tag{2-8}$$

The choice of the decoder network f_{dec} is again dependent on the type of the output data. When the output data is a sequence an RNN is a natural choice. In machine translation [9] an RNN is used to encode the input sentence. As encoder an RNN is used to predict the next word in the sequence. The encoding $\mathbf{c}$ is the representation from the last hidden layer, the layer before the softmax prediction over the next word (after the whole sentence was seen by the RNN). In some of the works on image captioning the output from the last convolutional layer is flattened to form the context $\mathbf{c}$. In other works the output from the last fully-connected layer before the output is used. The idea behind using the first option is that more spatial information is preserved. For both machine translation and image captioning an RNN language model is used as decoder.

2-4 Handwriting Synthesis

An application where recurrent neural networks and mixture density networks are used together for real-valued data is handwriting synthesis [2]. This is an interesting application, because it has similarities with generating manipulation trajectories for a robotic arm. For handwriting synthesis trajectories on a two-dimensional plane are generated. For robotic manipulation, trajectories in a three-dimensional space are generated. The handwriting is generated step by step, one pen offset per step. There are complex long range structures involved. For example, the network has to remember how to finish a character it has started a number of steps ago. Think of adding the dot on the 'i' or adding the cross for a 't'. By using a mixture density network there is a probability distribution for each predicted step. In this way there is a distribution over handwriting strokes. Furthermore, it can be shown that there is relative high uncertainty about where to start with a new character.

In conditioned handwriting, or synthesis, the input is some text (a sequence of characters). The output is a sequence of pen offsets, together with an indicator whether to lift the pen up. This represents handwriting. These two sequences are related. For this case no use is made of an encoder-decoder structure. Here this would not make sense, as in this application the input is not a rich high-dimensional structure. The RNN that generates the output strokes just needs to focus on one character of the input at the same time.

For the handwriting the basic RNN structure is the same as in Section 2-1. Each input x_t consists of a real-valued pair (x_1, x_2) that defines the pen offset from the previous input. The binary x_3 indicates whether this step is the end of a stroke (the pen is lifted up before the next stroke starts). A mixture of bivariate Gaussians was used to predict x_1 and x_2. For x_3 a Bernoulli distribution was used. There each output vector consists of the end of stroke probability e, a set of means μ^j, standard deviations σ^j, correlations ρ^j and mixture weights

π^j for M mixture components. That is

$$x_t \in \mathbb{R} \times \mathbb{R} \times \{0,1\} \quad , \tag{2-9}$$

$$y_t = \left(e_t, \{\pi_t^j, \mu_t^j, \sigma_t^j, \rho_t^j\}_{j=1}^M\right) \quad . \tag{2-10}$$

The probability density $\Pr(x_{t+1}|y_t)$ for the next input x_{t+1} given the output y_t is defined as:

$$\Pr(x_{t+1}|y_t) = \sum_{j=1}^{M} \pi_t^j \mathcal{N}(x_{t+1}|\mu_t^j, \sigma_t^j, \rho_t^j) \begin{cases} e_t & \text{if } (x_{t+1})_3 = 1 \\ 1 - e_t & \text{otherwise} \end{cases} \quad . \tag{2-11}$$

where $\mathcal{N}(x|\mu, \sigma, \rho)$ is the Gaussian indicating the probability of a x given mean vector μ, standard deviation vector σ and correlation ρ (x, μ and σ indicate two dimensional vectors).

This can be substituted in Equation (2-5). We can take the negative logarithm of the probability density to determine the sequence loss:

$$\mathcal{L}(\mathbf{x}) = \sum_{t=1}^{T} -\log\left(\sum_j \pi_t^j \mathcal{N}(x_{t+1}|\mu_t^j, \sigma_t^j, \rho_t^j)\right) - \begin{cases} \log e_t & \text{if } (x_{t+1})_3 = 1 \\ \log(1 - e_t) & \text{otherwise} \end{cases} \quad . \tag{2-12}$$

Then the derivatives of the weights with respect to the loss were computed and the network was trained on a dataset with handwriting examples with *rmsprop* [26]. Samples could be generated by iteratively sampling from the trained model. Some generated samples of this work are visible in Figure 2-3.

Note that this was the unconditioned case. For the case of handwriting synthesis, the generation of handwriting for a given text, an augmentation was needed. The main challenge is that the text sequence is much shorter than the pen trace and the alignment between them is unknown until the trace is generated. This is because the number of coordinates used to write each character varies and is also dependent on style and size of the handwriting.

The solution was to use a 'soft window', that convolves with the text string. This 'soft window' is fed as an extra input to the model. The parameters of the model are extra outputs, at the same time when predictions for the next step are made. So it dynamically determines an alignment between the text and the pen locations. It learns to decide which character to write next. Because we have no alignment problem for our case of generating trajectories — based on a vector representation of the task and environment — we do not go into detail on this 'soft window' model.

In terms of the loss function we now – for the case of handwriting synthesis — have to consider the text sequence $\mathbf{c}$ as well. The sequence loss is now

$$\mathcal{L}(\mathbf{x}) = -\log \Pr(\mathbf{x}|\mathbf{c}) \tag{2-13}$$

where

$$\Pr(\mathbf{x}|\mathbf{c}) = -\prod_{t=1}^{T} \Pr(x_{t+1}|y_t) \quad . \tag{2-14}$$

Note that y_t is now a function of the context vector $\mathbf{c}$ as well as the sequence of inputs $\mathbf{x}_{1:t}$.

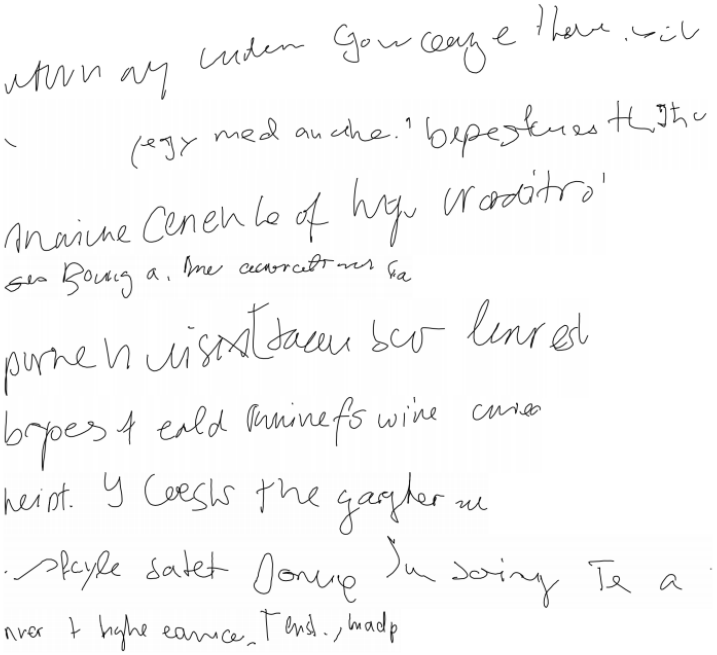


Figure 2-3: Handwriting samples generated by [2]. All samples are 700 steps long. Clearly characters and some shorts words are visible, this demonstrates the ability of the network to generate long-range structures, because the average character occupies more than 25 steps. Image from [2].

2-5 Current Approaches for the Robobarista dataset

For the application of handwriting the context $\mathbf{c}$ is a sequence of characters. For the task of trajectory generation we are interested in a context vector $\mathbf{c}$ representing the task and environment. A method to learn such a vector is described in [3]. The Robobarista dataset consists of point-clouds of an object to be manipulated, think of pushing a button or rotating a knob. Each pointcloud has a corresponding instruction in natural language, for example 'Push down on the handle to add hot water'. The dataset contains trajectories, collected via a crowd-sourcing platform, that perform the given instruction for the given object or appliance.

The goal in the Robobarista setting is to learn a function that maps a given new point-cloud $p \in \mathrm{P}$ of an object part and a new language instruction $l \in L$ to a trajectory $\tau \in \mathcal{T}$:

$$\mathrm{P} \times \mathrm{L} \rightarrow \mathcal{T} \quad . \tag{2-15}$$

In [3], this is done by learning a deep multimodal embedding. A deep neural network learns to embed the point-cloud/instruction pair to a feature space. For the manipulations trajectories an embedding to the same space is learned. For a new environment/instruction pair a trajectory can be selected by picking the nearest neighbor from the feature space.

This nearest neighbor trajectory often results in a reasonable performance. However, originally it was created with another (but quite similar) environment and instruction, which yields a trajectory that will not completely be customized for the new environment.

2-6 Comparison to Alternative Generative Models

In this chapter we described how to generate sequential data — handwriting and robotic manipulation trajectories are examples of this — with RNNs. In general three different approaches are used for generative modeling [27]:

- Generative Adversorial Networks (GANs) [28] pose the training process as a game between a generator network and a second discriminator network. The discriminator tries to classify examples as coming from the true distribution or the model distribution (generated by the generator). When the discriminator notes a difference between the two distributions, the generator adjusts its weights slightly to make the difference go away. In theory, at the end the generator exactly reproduces the true data distribution. The discriminator is then unable to find a difference.
- Variational Autoencoder (VAEs) [29] formalize the problem in a framework of probabilistic graphical models. A lower bound on the log likelihood of the generated data is maximized.
- Models like RNNs, as we described in this chapter, train a network that models the conditional distribution of every datapoint y_t (from which x_{t+1} is sampled) given previous datapoints $x_{1:t}$. This method is effective for data as handwriting and text [30], which are natural sequences. However also an image can be considered as a sequence, when the conditional distribution of every pixel is modeled given all pixels from the top and left. Here the RNNs run both horizontally and vertically over the image, instead of just in one dimension. This method is used in [31] to generate natural images.

Chapter 3

Methods

We approach the generation of trajectories in the context of sequence learning. The aim is to predict, given an internal representation of the steps taken so far, what the next step will be. We treat this predicted next step as a real step and use it to predict the next step again. By iteratively applying this method complete sequences of variable length can be produced.

A challenge is, that for predicting the next step in a sequence, looking at the current step only is not enough. The relevant information from the previous steps has to be taken into account. Therefore a main challenge is to capture this relevant information. For example, a trajectory for pushing down a handle consist roughly of several stages. The handle is approached, then there is some small push, a small movement in the opposite direction of the push is made to release the pressure on the handle and then the gripper moves away. Here the information from the previous waypoints is needed to determine what the current stage is and what the next waypoint should be.[1]

Another example is turning a knob, the RNN has to know in which direction (clockwise or counterclockwise) it is turning and how much it has already turned the knob. This is information that has to be extracted from the previous steps. This ability to extract and encode the relevant information from the sequence seen so far must be developed during the training phase. The network has to learn how to encode relevant aspects of the sequence seen so far in its hidden state.

In this chapter we describe how we use this method to generate manipulation trajectories with recurrent neural networks. We first look to the case where the generation is not conditioned on anything, letting the network freely hallucinate about trajectories. Next we describe the case where the generation is conditioned on a high-level representation of some task — a feature vector encoding of the environment and instruction in natural language [3]. In this way the trajectory generation is guided to perform a task.

[1] We could say the current direction of moving is possibly enough to determine the stage of the trajectory. However, to get fluid and robust trajectories, it is beneficial to have the history as well.

3-1 Unconditioned Generation

We build upon the architecture for handwriting generation [2] (Section 2-4) to generate trajectories. The key component of recurrent neural networks for sequence generation is to predict the next item in the sequence and then treat this item as if it was real. In [2] *mixture density networks* are used to predict an entire probability density function for the next item. When mixture density networks are combined with recurrent neural networks, the output distribution for the next step is not only conditioned on the current input (the current step), but on the history of previous inputs (the previous steps in the sequence).

3-1-1 Architecture

A data point in a manipulation trajectory is more complex than the 3-dimensional data point for the handwriting application described in Chapter 2. We assume manipulation trajectories consist of a sequence of waypoints. We use the trajectories from the Robobarista dataset [3]. Each waypoint $(g, \mathbf{t}, \mathbf{r})$ consists of a discrete gripper status $g \in \{\text{“open”, “closed”, “holding”}\}$, position $\mathbf{t} \in \mathbb{R}^3$ and orientation, expressed as a quaternion[2], $\mathbf{r} \in \mathbb{R}^4$ with respect to the origin. We use, similar to the handwriting work, the offset with respect to the previous data point as input for the translation and rotation. The gripper status is represented by three binary indicators $x_{\text{gripper}} = \{0, 1\}^3$, one for each of the three possible gripper options. So the input consists of the gripper status x_{gripper}, the translation offset and the rotation offset with respect to the previous waypoint, x_{trans} and x_{rot}:

$$x_t \in \{0, 1\}^3 \times \mathbb{R}^3 \times \mathbb{R}^4 \quad . \tag{3-1}$$

From the output layer of the recurrent mixture density network we use $y_{\text{gripper}} = (g_o)_t, (g_c)_t, (g_h)_t$ to indicate the probability for each of the three possible gripper statuses. For the real-valued data, the translations and rotations, we use two sets of mixtures of Gaussians, one to predict the translation and another mixture of Gaussians to predict the rotation with respect to the previous waypoint. Each mixture component has a weight π_t. The weights of the translational components π_t^j and the rotational components π_t^l sum both up to one. Each mixture component has a multi-dimensional mean μ and a covariance matrix Σ.

This is different than existing works on handwriting [2, 12], which use two dimensions per mixture component and thus can output standard deviations and correlations directly. We *cannot* use output entries directly as entries for the covariance matrix, because the covariance matrix has to be symmetric and positive definite. That is why we choose to let each mixture component output a lower triangular matrix L. Then the covariance matrix is build by $\Sigma = LL^T + \epsilon I$. In this way we have a symmetric and positive definite covariance matrix. So the output looks like

$$y_t = ((y_{\text{gripper}})_t, (y_{\text{trans}})_t, (y_{\text{rot}})_t) \tag{3-2}$$

with

$$(y_{\text{gripper}})_t = ((g_o)_t, (g_c)_t, (g_h)_t) \tag{3-3}$$

[2] We explain quaternions in Appendix A.

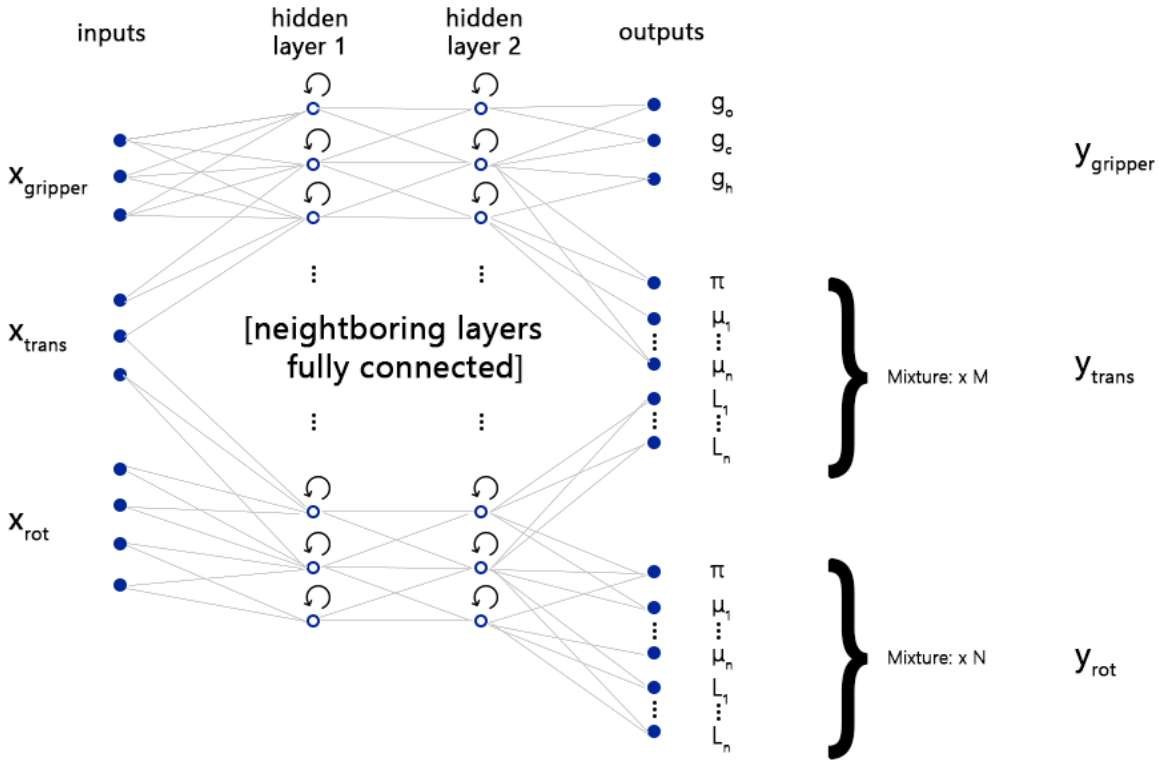


Figure 3-1: The architecture for the recurrent mixture density network.

where g_o, g_c, g_h indicate the probability for a open, closed and holding gripper status respectively. The other output terms y_{trans} and y_{rot} consist of mixture components:

$$(y_{\text{trans}})_t = \{\pi_t^j, \mu_t^j, L_t^j\}_{j=1}^M \quad , \tag{3-4}$$

$$(y_{\text{rot}})_t = \{\pi_t^k, \mu_t^k, L_t^k\}_{k=1}^N \tag{3-5}$$

with M and N the number of translation and rotational mixture components respectively. The architecture is depicted in Figure 3-1.

The softmax function is applied to let the each gripper status be in the $(0, 1)$ range and sum to one. To make the mixture weights for both the translational and rotational components sum to one also a softmax is applied:

$$\pi_t^j = \frac{\exp(\pi_t^j)}{\sum_{j'=1}^M \exp(\hat{\pi}_t^{j'})} \quad \Longrightarrow \pi_t^j \in (0,1), \sum_j \pi_t^j = 1 \quad . \tag{3-6}$$

Note that the means are three dimensional vectors for the translational and four dimensional for the rotational mixtures. To obtain corresponding 3×3 and 4×4 covariance matrices, the lower triangular matrices have 6 and 10 entries respectively. This is a lot more than in [2], where the mean and standard deviations are two dimensional vectors and there is one correlation number. When we compare the mixtures for trajectories with the mixtures for handwriting in [2], we go from a two-dimensional space without rotations involved to a three

dimensional space with rotations involved. This means going from 5 (2 means, 2 standard deviations and a correlation) numbers per mixture to 9 (3 means, 3 standard deviations and 3 correlations) per translational component and 14 (4 means, 4 standard deviations and 6 correlations) per rotational component.

With the output of the network — the gripper probabilities and the parameters of the mixture components for the translations and rotations — a probability density function can be formed. The next gripper status, next translation and next rotation are handled as independent components in our model, while they are strictly speaking not independent. This implies the output y_t contains independent predictive distributions for the gripper status, translation and rotation for x_{t+1}. This means that, for the example of releasing a handle, a closed gripper status could be sampled, while there is for step $t+1$ a translation in some direction sampled. Ideally the sampled gripper status should then be open to release the handle. However, this can already be fixed at the prediction for the next waypoint at step $t+2$, where the still closed gripper and the translation are provided as input. This is why this will not be a real problem, because the waypoints are close to each other. Because of the independence assumption, which makes the model much simpler than without, the probability $\Pr(x_{t+1}|y_t)$ of the next input x_{t+1} given the output of the network y_t is defined as:

$$\Pr(x_{t+1}|y_t) = \Pr_{\text{trans}}(x_{t+1}|y_t)\Pr_{\text{rot}}(x_{t+1}|y_t)\Pr_{\text{gripper}}(x_{t+1}|y_t) \tag{3-7}$$

$$= \Pr((x_{\text{trans}})_{t+1}|(y_{\text{trans}})_t)\Pr((x_{\text{rot}})_{t+1}|(y_{\text{rot}})_t)\Pr((x_{\text{gripper}})_{t+1}|(y_{\text{gripper}})_t) \tag{3-8}$$

with

$$\Pr((x_{\text{trans}})_{t+1}|(y_{\text{trans}})_t) = \sum_{j=1}^{M} \pi_t^j \mathcal{N}((x_{\text{trans}})_{t+1}|\mu_t^j, \Sigma_t^j) \quad , \tag{3-9}$$

$$\Pr((x_{\text{rot}})_{t+1}|(y_{\text{rot}})_t) = \sum_{k=1}^{N} \pi_t^k \mathcal{N}((x_{\text{rot}})_{t+1}|\mu_t^k, \Sigma_t^k) \tag{3-10}$$

and

$$\Pr((x_{\text{gripper}})_{t+1}|(y_{\text{gripper}})_t) = \begin{cases} (g_o)_t & \text{if } ((x_{\text{gripper}})_{t+1})_1 = 1 \\ (g_c)_t & \text{if } ((x_{\text{gripper}})_{t+1})_2 = 1 \\ (g_h)_t & \text{if } ((x_{\text{gripper}})_{t+1})_3 = 1 \end{cases} \quad . \tag{3-11}$$

The multidimensional Gaussian $\mathcal{N}(x|\mu, \Sigma)$ is defined as

$$\mathcal{N}(x|\mu, \Sigma) = \frac{1}{(2\pi)^{n/2}|\Sigma|^{1/2}} \exp\Big(-\frac{1}{2}(x-\mu)^T \Sigma^{-1}(x-\mu)\Big) \tag{3-12}$$

with Σ the covariance matrix built from the lower triangular matrix L with $\Sigma = LL^T + \epsilon I$.

As a loss function we take the negative log likelihood of the sequence, see Equation (2-5):

$$\mathcal{L}(\mathbf{x}) = \sum_{t=1}^{T} -\log \Pr(x_{t+1}|y_t) \tag{3-13}$$

$$= \sum_{t=1}^{T} \left[-\log \Pr_{\text{trans}}(x_{t+1}|y_t) - \log \Pr_{\text{rot}}(x_{t+1}|y_t) - \log \Pr_{\text{gripper}}(x_{t+1}|y_t) \right] \quad . \tag{3-14}$$

The loss function has a term for the translation, the rotation and for the gripper status. To give a term more importance with respect to the other terms we use weights. The respective terms are weighted with respect to each other with the α ($\alpha \geq 0$), β ($\beta \geq 0$) and γ ($\gamma \geq 0$) parameters:

$$\mathcal{L}(\boldsymbol{x}) = \sum_{t=1}^{T} \left[-\alpha \log \Pr_{\text{trans}}(x_{t+1}|y_t) - \beta \log \Pr_{\text{rot}}(x_{t+1}|y_t) - \gamma \log \Pr_{\text{gripper}}(x_{t+1}|y_t) \right] \quad . \tag{3-15}$$

If we substitute this backwards, the α, β and γ parameters corresponds to powers in the original probability $\Pr(x_{t+1}|y_t)$:

$$\Pr(x_{t+1}|y_t) = \Pr_{\text{trans}}(x_{t+1}|y_t)^{\alpha} \Pr_{\text{rot}}(x_{t+1}|y_t)^{\beta} \Pr_{\text{gripper}}(x_{t+1}|y_t)^{\gamma} \quad . \tag{3-16}$$

This means that when one of the weight parameters is increased this term get more influence on $\Pr(x_{t+1}|y_t)$.

3-1-2 Unbiased Sampling

A unbiased trajectory can be generated by iteratively sampling x_{t+1} from $\Pr(x_{t+1}|y_t)$, with t the time step ranging from 0 to some horizon T for a trajectory of $T+1$ waypoints. A mixture component is sampled for both the translational component and the rotational one, based on the weights π_t^j and π_t^k. After choosing the mixture components the translation and rotation are sampled from the distribution of the component itself, with the mean μ and Σ corresponding to that component.

3-1-3 Biased Sampling

When sampling from the output distributions to get real waypoints and trajectories, we can have a preference for smoother trajectories. In this case we should pick the mixture components with high probability more often. To bias the sampling towards more probable predictions at each step independently, we introduce the probability bias b. This is a real number greater than or equal to zero. The approach in [2] for recalculating the mixture weights is followed:

$$\pi_t^j = \frac{\exp(\pi_t^j(1+b))}{\sum_{j'=1}^{M} \exp(\hat{\pi}_t^{j'}(1+b))} \quad . \tag{3-17}$$

Each Gaussian mixture is scaled by

$$\Sigma_t = \frac{1}{1+b}\Sigma_t \quad . \tag{3-18}$$

This artificially reduces the variance in the choice of the mixture component and in the distribution of the mixture itself. When $b = 0$ the sampling remains unbiased, and when $b \to \infty$ the variance in the sampling disappears. In that case always the mean of the most probable mixture component will be chosen.

3-2 Trajectory Synthesis

In the previous section we described how to use a recurrent mixture density network to generate manipulation trajectories by predicting a probability density for the next waypoint. It is clear the generation of a trajectory *for a given task* is not possible in this way. There is no way to guide which movements the network generates, the networks freely hallucinates about trajectories. Here we describe an augmentation of the network that makes it possible to generate sequences conditioned on some high-level input.

3-2-1 Architecture

For the trajectory synthesis we use the learned compact representations from [3], see Chapter 4. This part acts as the encoder in our encoder-decoder structure. The learned representation acts as the context $\mathbf{c}$. It is an encoded form of the task description and the object part that has to be manipulated. The task of the RNN is the decode this context into a trajectory. The context $\mathbf{c}$ is connected to the first hidden layer of the RNN. The new architecture is depicted in Figure 3-2. If we denote the hidden state of the RNN as $\mathbf{h}_t$ we can update the hidden state of the RNN by

$$\mathbf{h}_t = \phi_\theta(\mathbf{h}_{t-1}, x_t, \mathbf{c}) \quad . \tag{3-19}$$

The context $\mathbf{c}$ for handwriting generation (Section 2-4) is a sequence of one-hot vectors. The dimension of each vector equals the number of characters in the alphabet. All vector entries are zero, except for the character which it represents. This entry equals one, therefore the name one-hot vector. Compared to this context for handwriting synthesis, we have one fixed size vector $\mathbf{c}$ with continuous entries for trajectory synthesis.

Because we use as inputs to the network the translation and rotation with respect to the previous point, instead of the origin, the network does not have any knowledge about the initial absolute configuration, the initial position and orientation. That is why we add the initial position and orientation to the context vector $\mathbf{c}$.

Another option, which is similar to the work on generating handwritten digits [12], is to provide the initial position and orientation as the first waypoint and let the network generate the translations and rotations for the next waypoints. This is similar to [12], because there the first point provided is the offset from the origin. In the handwriting work the top-left corner of the canvas is the origin, where we assume the origin to be in the object part to be manipulated. This approach seems not ideal, distances from the origin can be quite large and are not in line with the other translations. This could confuse the network.

If we follow the simple encoder-decoder framework [18] we should initialize the hidden state of the RNN with the context vector $\mathbf{c}$. The problem here is that there is a mismatch between the size of the context vector and the hidden state of the RNN. However, the remaining units in the hidden layer could still be initialized to zero values.

For analysis of the contexts we use a 2-dimensional t-SNE [32] embedding to visualize the context vectors $\mathbf{c}$. In this way we can get an intuition for what is encoded in these vectors. We expect vectors for similar instructions to appear together.

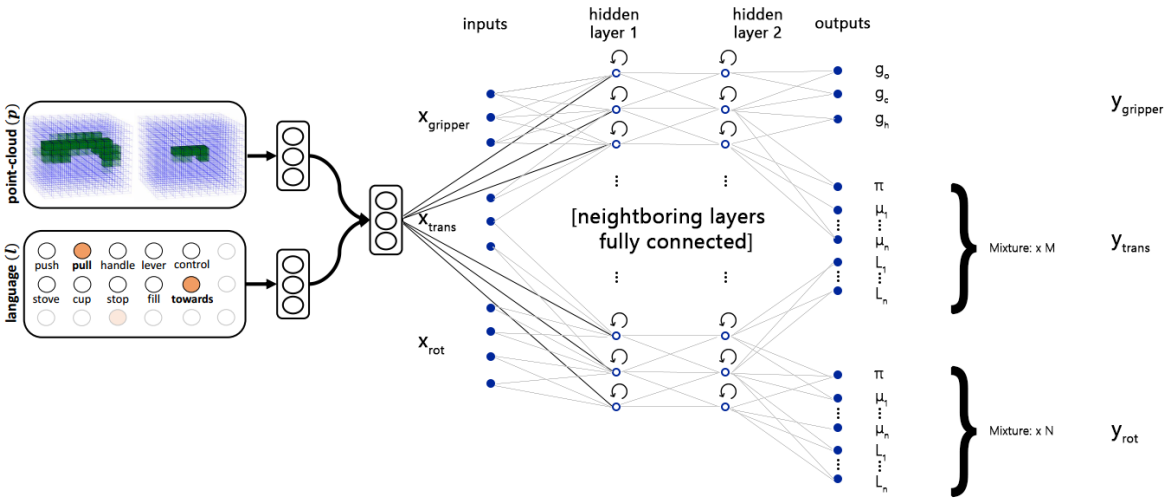


Figure 3-2: The architecture for the recurrent mixture density network, conditioned on a feature representation of an instruction in natural language and a point-cloud of the object part that has to be manipulated. The feature representation is added as extra input to the first hidden layer.

Chapter 4

Encoder

We build a neural network that projects input pairs — consisting of a point-cloud of an *manipulatable* object part and an instruction in natural language, saying what to do with the object part — to a feature space. The goal in later stages is to use these learned feature representation to generate new trajectories from. We choose to reproduce the method described in [3], which achieves state of the art performance on the Robobarista dataset [19].

4-1 Preprocessing

The raw data from the three different modalities — point-clouds, natural language instructions and trajectories — are converted to fixed-length vector representations. This fixed length vectors are the inputs to the networks, one network for mapping a pointcloud-language pair to the feature space and another network to map trajectories to the same feature space.

4-1-1 Point-cloud

The raw RGB-D information of the segmented point-cloud — only the part of the point-cloud containing the object part is included — is converted to two occupancy grid like structures. Both structures have $10 \times 10 \times 10$ voxels. One structure has a voxel size of $1 \times 1 \times 1$ cm. The other structure has a voxel size of $2.5 \times 2.5 \times 2.5$ cm. We do not take into account the color information in the raw RGB-D data, but only the position information. Each voxel counts how many points of the point-cloud are contained in that voxel. The grid structure with smaller size has more precision, but not all points are contained in this smaller sized structure. The bigger structure does not miss points, but is less accurate. With two $10 \times 10 \times 10$ structure the point-cloud is converted to a 2000-dimensional vector representation. Both grids are normalized to contain values between 0 and 1 by dividing by the maximum count in the grid.

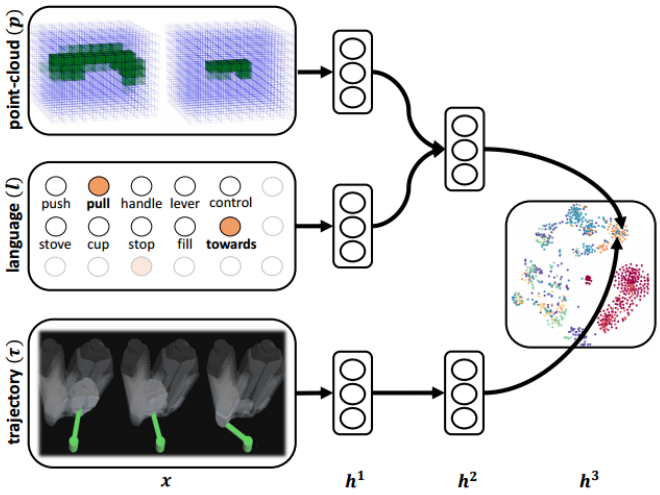


Figure 4-1: The model for the deep multimodal embedding. Figure from [3].

4-1-2 Natural language instruction

For the language instruction a bag-of-words model is used. The total number of different words contained in the dataset is 230. So each instruction is converted to a 230-dimensional vector, where each vector element is set to 1 if that word is contained in the instruction and to 0 otherwise.

4-1-3 Trajectory

Trajectories are given as a variable-length sequence of waypoints. Each waypoint contains a gripper status ('open', 'closed' or 'holding'), a translation with respect to the origin and a rotation (represented as a quaternion) with respect to the origin. Each trajectory is normalized to a length of 15 waypoints by removing waypoints or adding them through interpolation. For each waypoint we have 3 binary features for the gripper status, 3 real-valued ones for the translation and 4 real-values ones for the quaternion. This yields a 15×10 dimensional feature representation for each trajectory.

4-2 Network architecture

We have two separate networks. One network $\Phi_{P,L}(p, l)$ has as input the point-cloud representation p and the instruction representation l. The other network $\Phi_{\tau}(\tau)$ has as input the fixed-length representation of the trajectory τ. Both networks have as output an M-dimensional layer. This are the M features we want to learn. So both networks project their input to the shared feature space. The architecture of the network is depicted in Figure 4-1. This is the same architecture as in [3]. The number of neurons in the layers $h_{1,p}$, $h_{1,l}$, $h_{1,\tau}$, $h_{2,pl}$ and $h_{2,\tau}$ is respectively 150, 175, 100, 100 and 75. This is the same as in [3].

4-3 Loss Function

We want to learn *useful* features. This means features that could be used to generate trajectories from. In standard auto-encoder networks the objective is to reconstruct the original input. Then the representation in a hidden layer — often with a small number of neurons to "compress" information — could serve as feature representation. In this case we do not use this reconstruction objective as loss function, but use the same loss function as in [3].

This loss function is constructed by looking more specifically at the problem. For each point-cloud/language pair (p_i, l_i) some trajectories should be able to complete the task, while others are completely irrelevant. We want the relevant trajectories to be close in the feature space to the pair (p_i, l_i), while the irrelevant trajectories should be further away. To express similarity in the feature space a similarity function is introduced, similarity is defined as $\text{sim}(a, b) = a \cdot b$. The relevant trajectories should have a higher similarity to the projection of (p_i, l_i) than the irrelevant trajectories. Besides that, we want that some irrelevant trajectories are more irrelevant than others. To be more specific, the difference between the similarities of τ_j and τ_k to the projected point-cloud/language pair (p_i, l_i) should be at least the loss $\Delta(\tau_j, \tau_k)$. Here $\Delta(\tau_j, \tau_k)$ is a loss function that compares demonstrations. So we have the constraint

$$\text{sim}(\Phi_{P,L}(p_i, l_i), \Phi_\tau(\tau_j)) \geq \Delta(\tau_j, \tau_k) + \text{sim}(\Phi_{P,L}(p_i, l_i), \Phi_\tau(\tau_k)) \tag{4-1}$$

where τ_j is a relevant trajectory for the pair (p_i, l_i) and τ_k is an irrelevant trajectory.

To determine the set of relevant and irrelevant trajectories the optimal trajectory is determined for each pair (p_i, l_i). The trajectories are collected using crowd-sourcing. We assume the trajectory with the smallest average distance to all other trajectories for the pair (p_i, l_i) must be a good demonstration. We use this as the optimal demonstration τ_i^* and use thresholds (t_S, t_D) to determine two sets of relevant (similar) and irrelevant (dissimilar) trajectories from the total pool of trajectories T:

$$\begin{aligned} T_{i,S} &= \{\tau \in T | \Delta(\tau_i^*, \tau) < t_S\} \\ T_{i,D} &= \{\tau \in T | \Delta(\tau_i^*, \tau) < t_D\} \end{aligned} \tag{4-2}$$

The values used for t_S and t_D are 8 and 10. The dataset grows exponentially if we push away all irrelevant trajectory at each iteration for each pair (p_i, l_i). Instead, the *most violating trajectory* τ' can be determined. This is the trajectory from the set of irrelevant trajectories with the highest similarity augmented with the loss scaled by a constant α:

$$\tau_i' = \text{argmax}_{\tau \in T_{i,D}}(\text{sim}(\Phi_{P,L}(p_i, l_i), \Phi_\tau(\tau_j))) + \alpha\Delta(\tau_i, \tau)) \tag{4-3}$$

The value used for α is 0.2. So at each iteration for each example (p_i, l_i) we want to push away the most violating trajectory τ_i'. Therefore we can use the hinge loss of the most violating trajectory as a loss function:

$$L(p_i, l_i, \tau_i) = |\Delta(\tau_i', \tau_i) + \text{sim}(\Phi_{P,L}(p_i, l_i), \Phi_\tau(\tau_i')) - \text{sim}(\Phi_{P,L}(p_i, l_i), \Phi_\tau(\tau_i))|_+ \tag{4-4}$$

Here the notation $|\cdot|_+$ is the same as $\max(0, \cdot)$. This loss function equals 0 when the relevant trajectory has a similarity equal or greater than the similarity of the irrelevant one augmented with the distance between the trajectories.

4-4 Experiments

The model is trained on the Robobarista dataset consisting of 250 point-cloud/language pairs and 1225 crowd-sourced manipulation trajectories. We use roughly 80% as training set and the remaining 20% as test set. We use the Adam optimizer [33] with an initial learning rate of 0.001, a small decay is applied after every $10,000$ steps. We used the same parameters for t_S, t_D and α as in [3], values of 8, 10 and 0.2 respectively. From initial experiments we found that using all dissimilar trajectories, instead of only the most violating ones, gives better performance. This leads to large dataset of more than 18 million combinations of pointclouds, instructions, similar and dissimilar trajectories. The general opinion in deep learning is that more data is beneficial, so this could explain why this works better. Although by using all dissimilar trajectories, we also include a large number of trajectories that are already far enough away from the pointcloud/language pair.

We used a batch size of 256 and trained the networks for $20,000$ iterations. The loss for the model is shown in Figure 4-2. We also test the number of constraints (Equation 4-1) satisfied, this is shown in Figure 4-3. We see a large gap between the training and test set.

We experimented with using Batch Normalization to improve learning. However, we found batch normalization is not suited for this task. By using batch normalization, activations of units depend on the other input examples in the batch. When an input example consists of a pointcloud/language pair, a similar trajectory and a dissimilar trajectory $(p_i, l_i, \tau_j, \tau_k)$, this means the similar and dissimilar trajectory τ_j and τ_k have different mean and variance values per unit. Therefore the same trajectory can have different embeddings. Therefore the training loss will decrease very fast, it can "cheat" by using different embeddings for the same trajectory and by using that satisfy constraints. This results in a model that is performing bad during inference, when the mean and variance are fixed. The mean and variance computed during training are used then.

Batch renormalization [34] claims to ensure that training and inference models generate the same outputs that depend on individual examples, rather than on the entire minibatch. Therefore batch renormalization could be a suitable option to improve the performance of the model.

In the rest of this work we do not make use of this results, because we are able to use the original pointcloud/language pair embeddings used in [3].

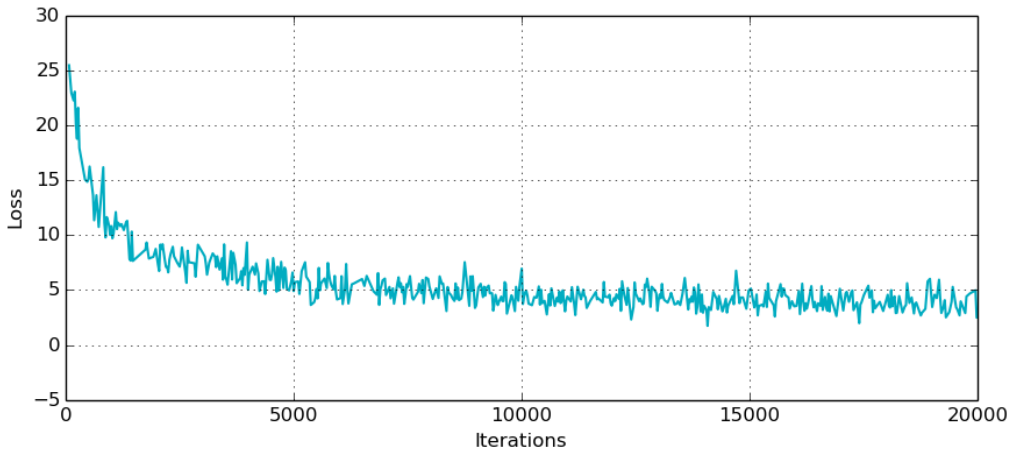


Figure 4-2: The loss during training.

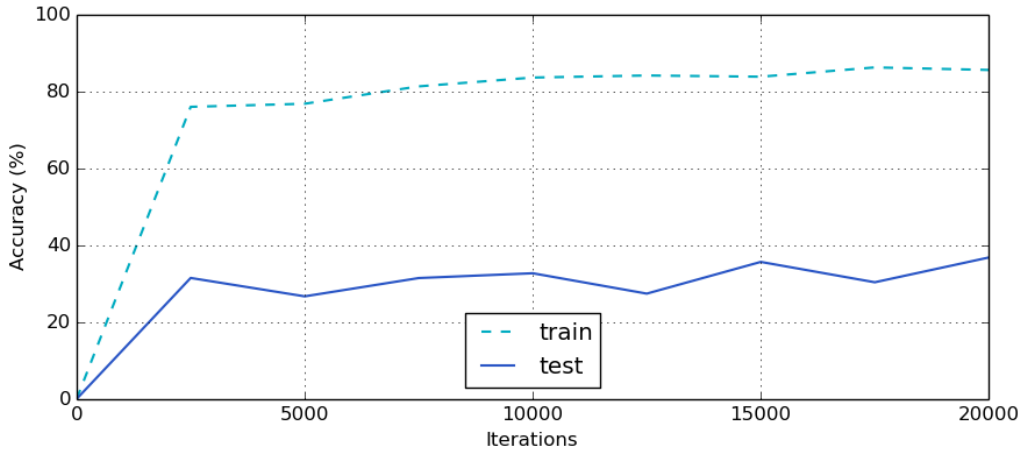


Figure 4-3: The percentage of constraints satisfied during training. We see a gap between the train and test set.The model performs much better on the training set, which is a sign of overfitting and not generalizing well to new data.

Chapter 5

Experiments

In Chapter 3 we described our model that predicts the next waypoint based on an internal representations of the waypoints seen so far.

We train the model on the Robobarista dataset [19]. This dataset contains 1225 manipulation trajectories which are demonstrated for 250 tasks. We use roughly 80 percent for training the model and use the remaining data to report performance on.

5-1 Dataset

The Robobarista dataset [19] contains three different modalities of data (point-clouds, language instructions and trajectories). There are 116 objects in the dataset with 250 natural language instructions. Using crowd-sourcing 1225 manipulation trajectories are collected from 71 non-expert users. The trajectories are collected via a web platform (http://robobarista.cs.cornell.edu). Because of the collection via crowd-sourcing the trajectories contain a lot of noise.

We note that the number of pointcloud/language pairs and trajectories is relatively small. In the slightly comparable application of generating handwriting more than $10,000$ handwritten lines are used, with an average line occupying around 700 steps (the average letter about 25 steps. This is a lot more than the robotic trajectories we have available. A lot of trajectories do contain a minimal amount of waypoints, some trajectories contain only three or four waypoints. This low resolution of waypoints can result in different trajectories for the same task that look quite different, which makes it very challenging to capture the relevant patterns in the training phase. The adding of extra waypoints via linear interpolation does not remove this noise in the training trajectories.

Each point cloud $p \in \mathrm{P}$ is represented as a set of n points in a three-dimensional Euclidean space. Each point (x, y, z) is represented with its color (r, g, b) resulting in:

$$p = \{p^{(i)}\}_{i=1}^{n} = \{(x, y, z, r, g, b)^{(i)}\}_{i=1}^{n} \tag{5-1}$$

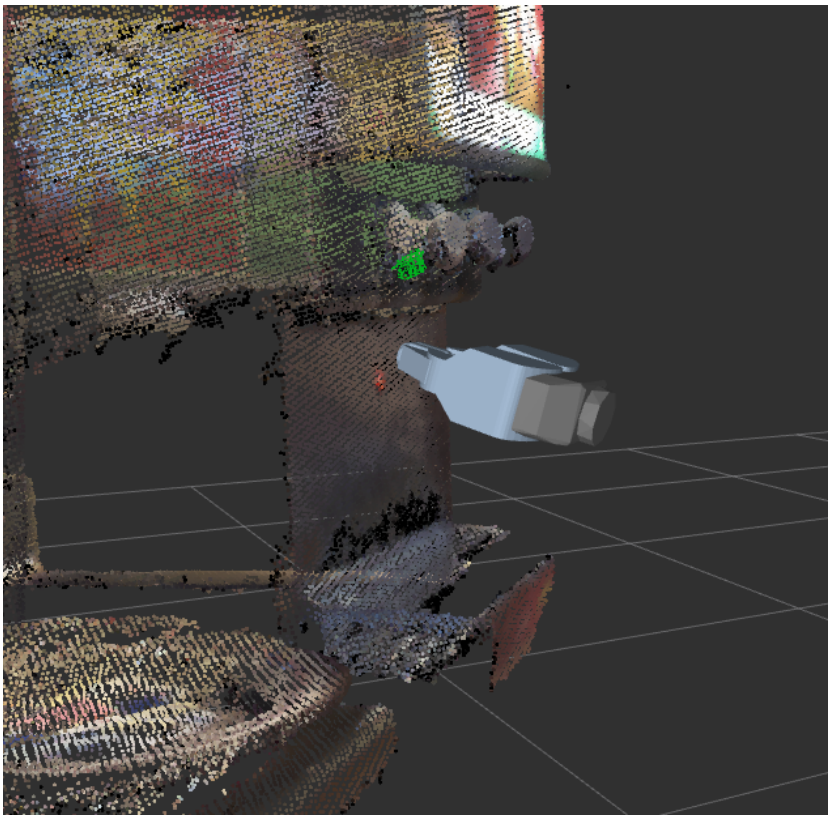


Figure 5-1: A point-cloud for a coffee cream dispenser. The corresponding instruction is 'Hold the cup below the nozzle'. Also shown is the two-fingered end-effector of the robot arm. The segmented object part, shown in green, is the only part that is used. The other points in the point-cloud are not used.

for a point-cloud with n points, see Figure 5-1. Each instruction $l \in L$ is written in natural language. An example is (for two instructions): *"Hold paper towel in front of the nozzle. Pull the trigger to spray."*. Here the task is to spray cleaner to a paper towel. Each trajectory $\tau \in \mathcal{T}$ is a robot arm trajectory and is represented with a sequence of m waypoints. Each waypoint consists of a gripper status $g \in \{$"open", "closed", "holding"$\}$, translation (t_x, t_y, t_z) and rotation (r_x, r_y, r_z, r_w) with respect to the origin:

$$\tau = \{\tau^{(i)}\}_{i=1}^{m} = \{(g, t_x, t_y, t_z, r_x, r_y, r_z, r_w)^{(i)}\}_{i=1}^{m} \tag{5-2}$$

The rotations are expressed as quaternions.

Only the position and rotation of the end-effector of the robot arm are recorded. In this way, focus is on the interaction between the robot and the environment, rather than the movement of the arm. Besides that, the trajectories are represented in an coordinate frame of the object part rather than the robot or object coordinate frame. This makes trajectories more easily compatible across different objects. The negative z-axis is aligned along gravity. The x-axis is aligned following the approach in [35], along the principal axis of the manipulatable object part using PCA. This means that when the robot is in front of the object — and wants to approach it — it has to move along the y axis in positive direction. By using this coordinate frame, the trajectory does not need to change, even when the object part's position and orientation changes. However, the approach of [35] does not always succeed. There are examples in the dataset where the end-effector has to move along the x-axis or diagonally.

5-2 Preprocessing

All trajectories are normalized to a fixed length of 25 waypoints. In this way the trajectories become smoother. The interpolation is done by adding extra waypoints via interpolation, linear interpolation for the positions and spherical interpolation for the rotational quaternions This is the same preprocessing step as for learning a multimodal embedding in [3]. The choice for 25 waypoints is a trade-off. Choosing a lower number results in less smooth trajectories, which makes it more difficult for the neural network to learn patterns. Choosing more waypoints per trajectory will result in an even greater percentage of interpolated waypoints. We do not want a network that only learns to interpolate and is not learning the more relevant patterns in the trajectories. An alternative for the fixed-length, which allows variable-length sequences, is to use an end-of-sequence marker. Because of ease of implementation, we decided to stick with the fixed-size version.

In the Robobarista dataset each trajectory consists of a sequence of waypoints. Each waypoint consist of a gripper status, a position and an orientation of the gripper with respect to the origin. We choose, similar to [2], to let the network output the translation and rotation with respect to the previous waypoint. This instead of predicting the absolute coordinates with respect to the origin. This makes the predictions depend on local features. This is more intuitive than when global coordinates have to be remembered and predicted.

A translation between two consecutive absolute positions can be computed in a straigthforward way, by subtracting the position at step t from the position at step $t+1$. The rotation between two consecutive quaternions q_1 and q_2, both representing the orientation with respect to the origin can be computed with

$$q' = q_2 \cdot q_1^{-1} \tag{5-3}$$

with the inverse and multiplication operator as defined for quaternions. Now we can go from orientation q_1 to q_2 by $q_2 = q' \cdot q_1$.

We have to note that by using the translations and rotations with respect to the previous waypoint, instead of absolute positions and orientations, we loose information about the initial position and orientation with respect to the origin. The same goes for the initial gripper status. For the gripper status this is not a problem, because the gripper status for the second waypoint turns out to be always equal to the first. It does not make sense to immediately change the gripper status. By loosing information about the absolute initial position and orientation, this means the preprocessed trajectories become translation and rotation invariant. Two trajectories with different initial positions and orientations, but performing the same motions (translations and rotations), will result in two identical preprocessed trajectories. Because of the loss of the original position and orientation, this is added to the context vector when we experiment with conditioned trajectory generation.

All translation and rotation features are normalized to have mean 0 and standard deviation 1, as is recommended in literature [36]. Because the gripper status consists already of binary values, these features do not have to be normalized. We note that by normalizing the rotation features, it is not a valid quaternion with unit length anymore. However, a 4-dimensional vector at the output is easily scaled to have unit length again [37].

5-3 Metrics

We train the network to predict the next step in the sequence — whether we provide the context vector $\mathbf{c}$ or not. A natural choice would be to consider the root-mean squared error (RMSE). When given a target sequence this network is given the true step at time t and asked to predict the step at $t+1$. The RMSE is the error between the predicted estimate for $t+1$ and the true value at $t+1$. This error is averaged over the total number of predicted steps.

However, we are not per se interested in the per step prediction capabilities. This is because we are interested in the prediction of whole trajectories. The network is trained with the teacher-forcing method. This means the ground-truth input at step t is given and the network is asked to predict the next step $t+1$. This prediction is not used as input at the next step, but again the ground truth value at time $t+1$ from the target trajectory is provided (therefore the name teacher-forcing) to predict step $t+2$.

We want that the predicted trajectory during training — during training no hallucinating is involved — matches the target trajectory as close as possible. That is why we make use of the metric Dynamic Time Warping for Manipulation Trajectories (DTW-MT) [19]. This is a metric that computes a loss between two complete trajectories by non-linearly warping the trajectories of possible different lengths (although we use fixed length in our implementation). With this metric translations, rotations and gripper status are taken into account. The ordering of matched trajectory waypoints is preserved.

Because the DTW-MT metric does not give an intuitive number, we also measure accuracy. By accuracy we mean the percentage of the trajectories with an DTW-MT value less than 2. By empirical evaluation, a value less than 2 from the ground-truth trajectory looks very similar. This accuracy metric is also used in [3] to evaluate the performance of their method.

In their work, this is not used in the context of teacher-forcing and thus DTW-MT values are higher. Therefore they use a threshold value of 10 to compare trajectories.

5-4 Trajectory Generation

First we look to the experiments where we do not provide the context $\mathbf{c}$. We train the model and then sample a number of trajectories from this trained model — by letting the model "hallucinate". The goal of this experiment is to see if the network has learned patterns from the trajectory data and is able to generate reasonable trajectories.

5-4-1 Training

We train using the Adam [33] algorithm for 100 epochs, using a batch size of 20. The learning rate starts at 1×10^{-3} with a small decay applied after every epoch. The derivatives are clipped in the range $[-10, 10]$.

We use 2 hidden LSTM layers [24] with each 256 units and 5 mixtures for both the translational and rotational components. This results in a network with roughly $830,000$ trainable parameters. This is in line with the different architectures used for online handwriting [2, 12].

We have to find the parameters α, β and γ to weight the translation, rotation and gripper term in the loss function and find a good balance between them. Training runs with equal weights for the translational, rotational and gripper term showed a much higher value for the gripper loss. Therefore we train a model with only the gripper term involved, thus paying no attention to the translation and rotation. In this way, we can find out what the optimal value for the gripper loss is. Because the model is trained with teacher-forcing — at each step the input from the target trajectory is provided — there are reasonable translations and rotations provided to base the prediction for the gripper on. The result for the gripper loss is shown in Figure 5-2.

Training runs with the equal weights for the translational, rotational and gripper term showed a much higher value for the gripper loss. Therefore we choose values for α, β and γ of 1, 1 and 100 respectively. In this way the gripper loss is close to the optimal value. The total loss during this training is plotted in Figure 5-3. We also look to the individual contributions of the translational, rotational and gripper term to the loss. Initially the gripper term has most influence, but this contribution decreases fast. To show that the learned predictions also works for trajectories never seen before — the validation set — we plot performance on the RMSE and DTW-MT measure in Figures 5-4 and 5-5.

The size of the dataset is quite limited, so we should expect to see signs of overfitting. The performance on the training data is better than performance on the test data, on the metrics reported here. When there is overfitting we could solve this by adding drop-out [38] to our network. A keep probability of 0.8 is used in work on handwriting [39]. To see the effect of this added drop-out we also train a model with drop-out. We choose a keep probability of 0.8. The results are in Figures 5-6, 5-7 and 5-8. We see the model with dropout performs better than the model with no drop out. However, as we will describe later in this chapter, a model with better predicting capabilities for the next waypoint will not generate better trajectories per se.

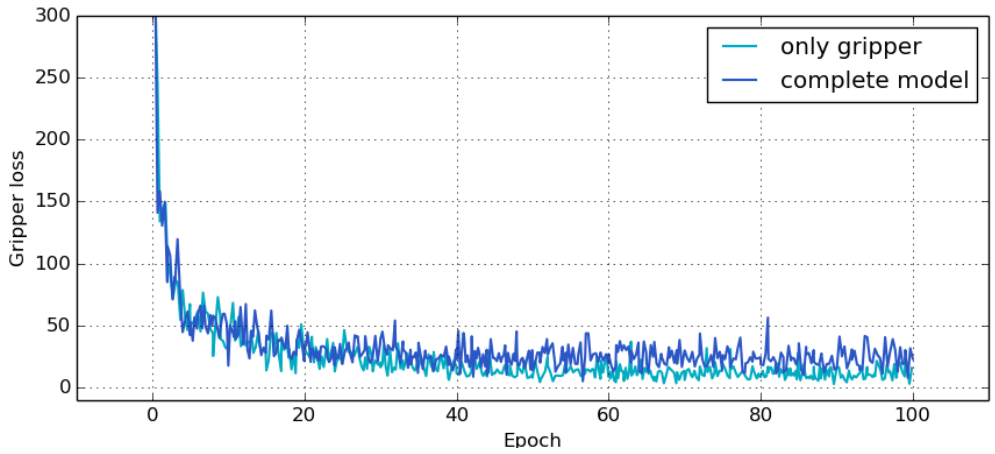


Figure 5-2: The value of the gripper term of the loss function during training. We compare a model that is trained for 100 epochs with only the gripper term in the loss function. From the other model, with a complete loss function we take only the contribution of the gripper term. The model fully trained on the gripper has a slightly lower loss. However, the difference is very small.

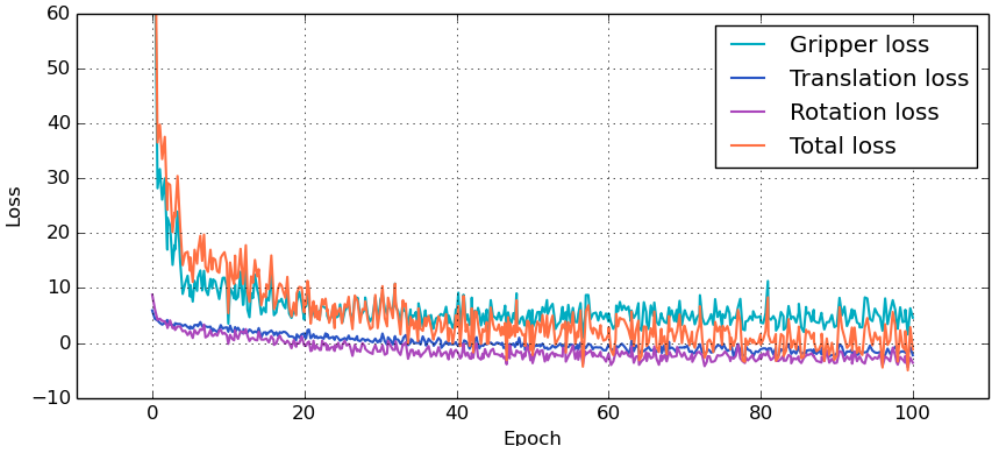


Figure 5-3: The loss on the training data. The network is trained for 100 epochs with a small decay applied to the learning rate after each epoch. Also shown are the individual contributions of the translation, rotation and gripper term to the loss. The gripper term is weighted by a factor of 100.

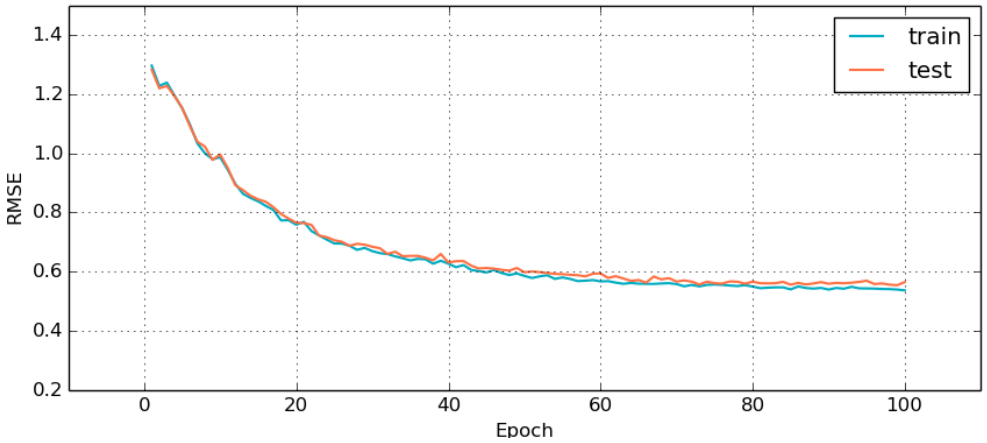


Figure 5-4: The root-mean squared error (RMSE) for both the training (blue) and test (orange) data. The RMSE is only computed for the predicted translations, so predicted rotations and gripper status are not taken into account.

To check what kind of predictions the network has really learned we visualize some trajectories from the validation set. Here we can see the predicted movements at each step. Because we use a mixture density network the network outputs multiple options for each step. The results are in Figure 5-9.

We see that there is always an option — for a model doing well on the training metrics — to do the same translation at step $t + 1$ as at step t. In general — due to the high number of interpolated waypoints in the trajectories — this is a good prediction. However, always predicting to same translation as the previous step can be considered as a sign of overfitting. This will probably not lead to good trajectories when generating complete trajectories with no target trajectory involved.

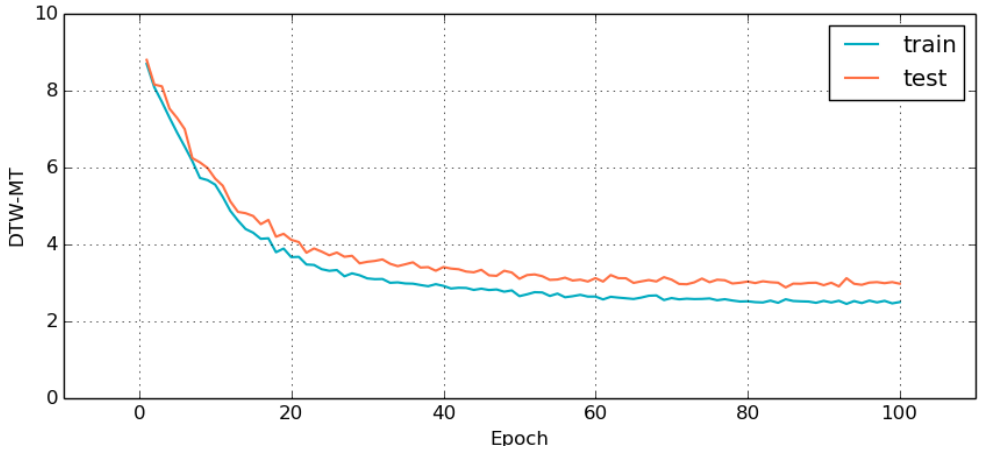
In Figure 5-10 and 5-11 we see another example. Here it is clearly shown that at the first step the model does not really have an idea in which direction to move. This is logical because it has not yet any information available. After some movements the model predict options with higher probabilities. This are most often movements in approximately the same direction.

5-4-2 Generated Trajectories

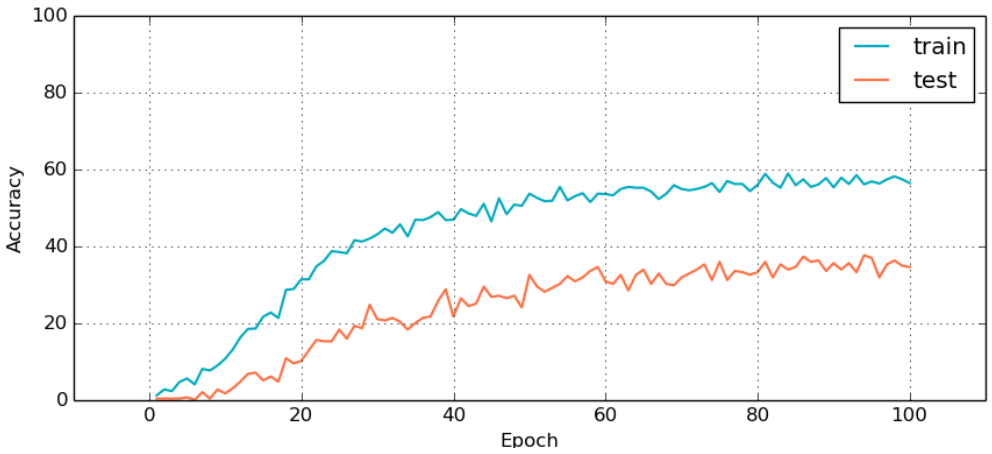
We sample trajectories from the model and then look at the results. There is no metric to measure the quality of hallucinated trajectories. We choose the model that gives best results, when looking at the visualizations. This is not the model with dropout, that performs best on the training metrics.

For hallucination the first waypoint is fed and the remaining waypoints are sampled from the model. The first waypoint consist of zero values for the translation and a unit quaternion for the rotation (meaning zero rotation). We experiment with different initial values for the gripper.

We expect that a lot of changes in gripper status between consecutive waypoints does indicate a bad trajectory and more consistency — only a few changes — indicates a better trajectory.



(a) DTW-MT



(b) DTW-MT Accuracy

Figure 5-5: DTW-MT plots showing results of the DTW-MT metric. The predicted waypoints are connected to form a trajectory. The loss between this trajectory and the target trajectory is computed with the DTW-MT metric. In this metric positions, rotations and gripper status are taken into account. The results shown here are averaged over all trajectories in the train or test set. Because the DTW-MT metric does not give an intuitive number we also plot the percentage of trajectories with a DTW-MT loss less than 2.

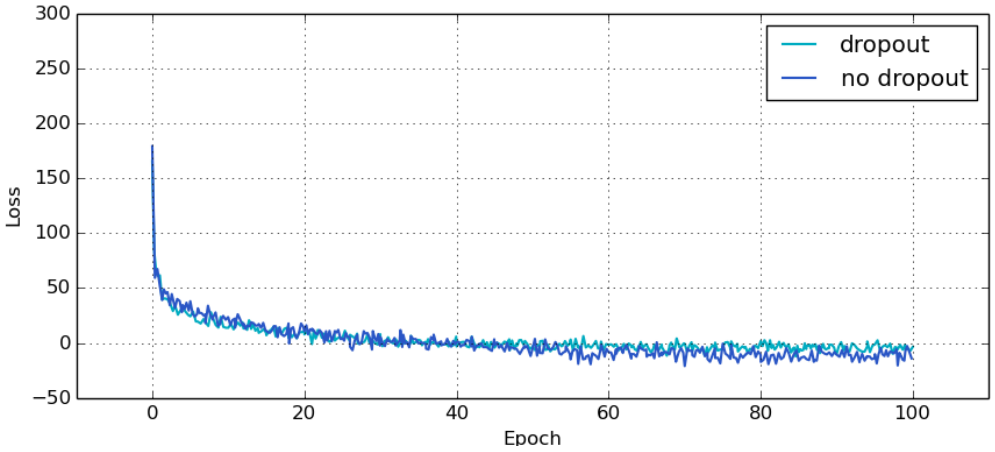


Figure 5-6: The loss on the training data for a network trained with and without dropout. We see the no dropout version achieves a slightly lower loss. This is explainable, with no dropout no units are dropped.

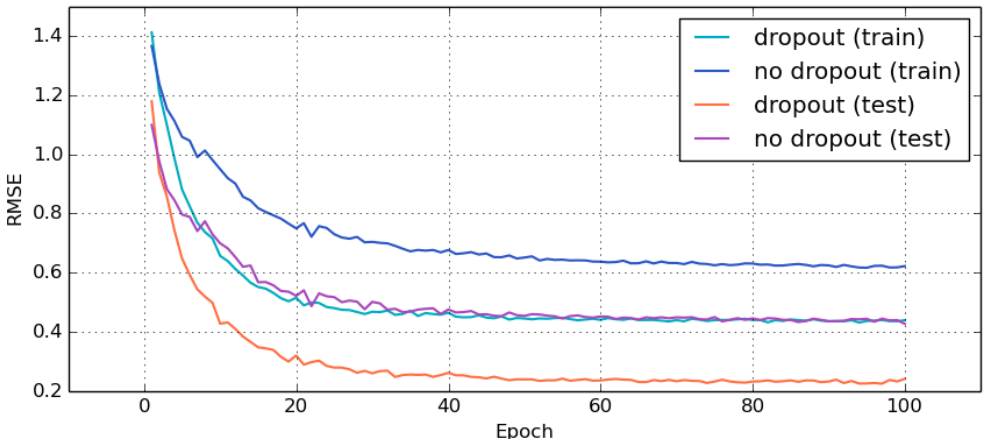
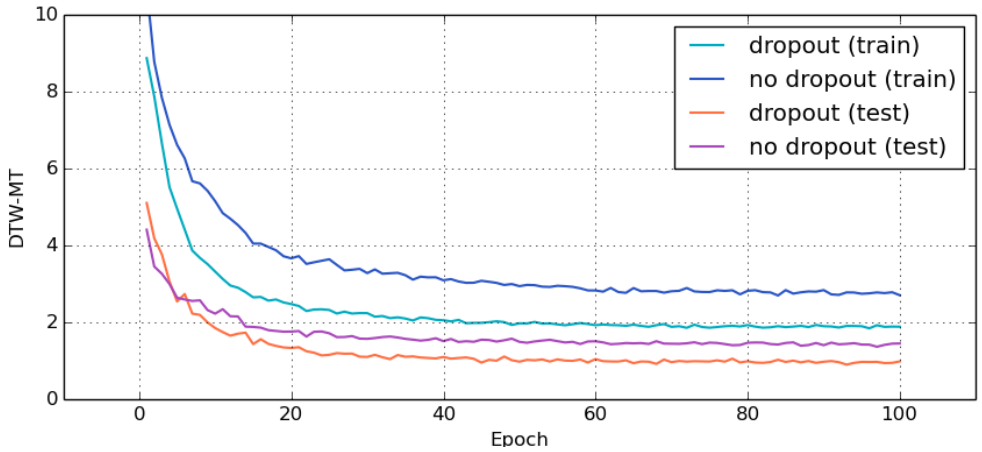
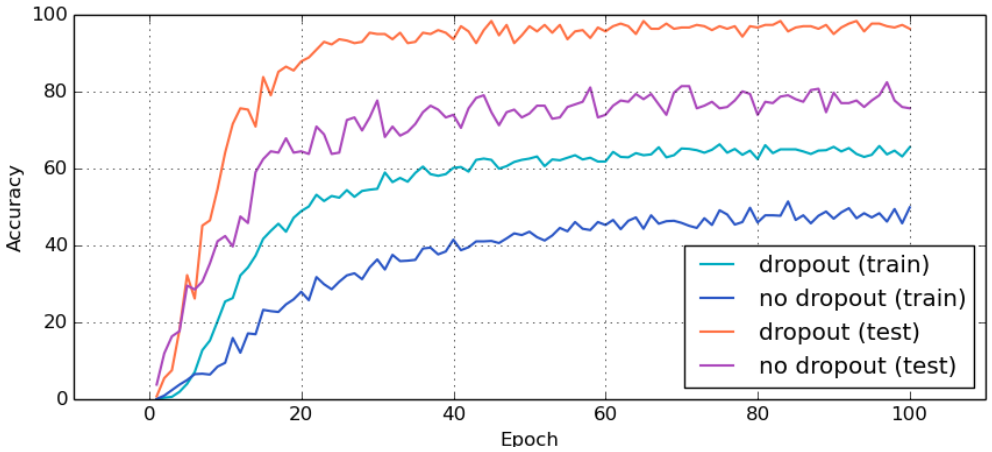


Figure 5-7: The root-mean squared error (RMSE) for both the training and test trajectories with and without dropout. During evaluation for both the training and test set no dropout is applied, no units are dropped. Remarkable is that the test trajectories perform better than the training trajectories. Also dropout performs clearly better than no dropout.

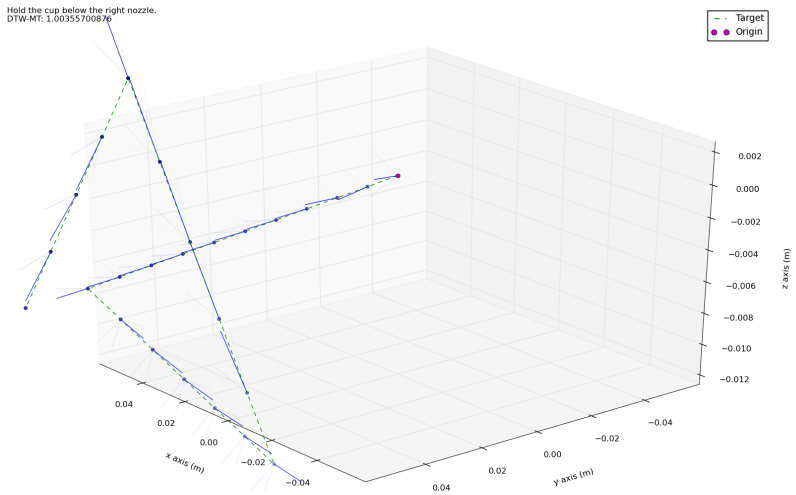


(a) DTW-MT

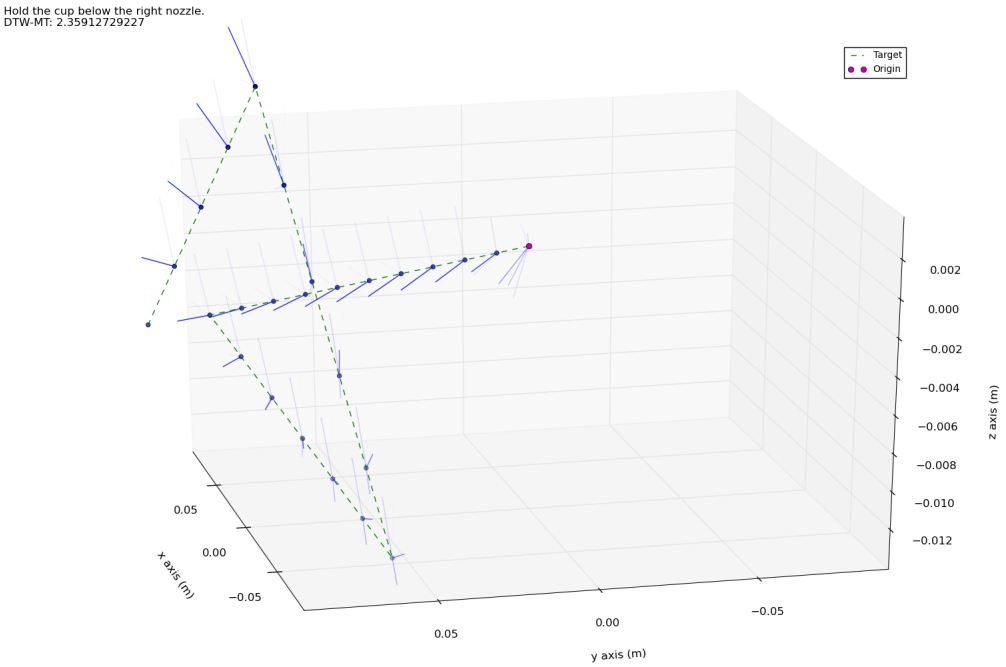


(b) DTW-MT Accuracy

Figure 5-8: DTW-MT plots showing results of the DTW-MT metric for dropout and no dropout. This gives the same view as the RMSE measure. Dropout performs better than no dropout. Remarkable is that the test trajectories perform better than the training trajectories.



(a) Example from model with dropout.



(b) Example from model without dropout in earlier stages of training.

Figure 5-9: Training visualized for a trajectory from the test set, only predicted translations are shown. The target trajectory is the dashed trajectory. At each waypoint the model is asked to predict the next move (teacher-forcing). The means from the mixture components are plotted (higher probability means higher opacity). We see that for this noisy trajectory the upper model achieves a better DTW-MT score than the lower one. However, the upper one prefers to keep moving in the same direction and misses the changes in direction. The lower one does not simply repeat the previous move, but keeps a preference for a move in a certain direction which slowly changes.

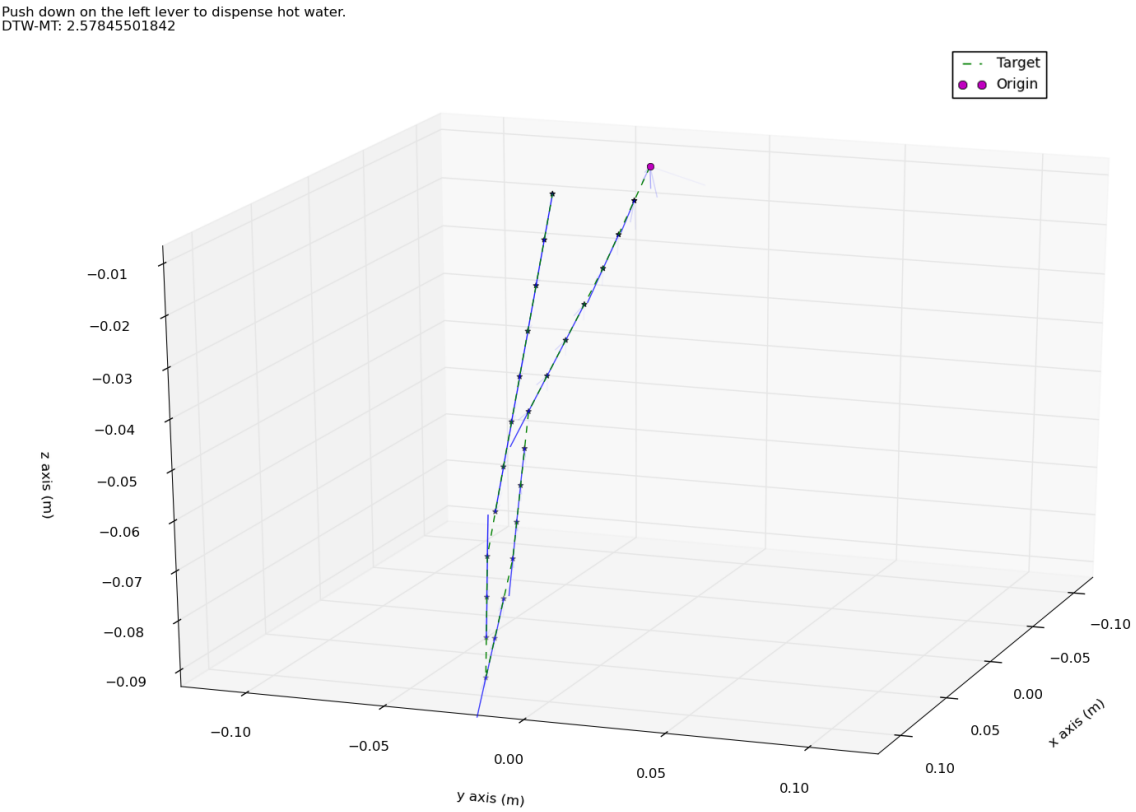


Figure 5-10: Typical example of a 'push down' trajectory to pull down a lever or handle. The object part is approached. A small push down movement is followed by a move in opposite direction and the closed gripper moves away again. We see that at the first step there is a high uncertainty in the model. All options have about equal probability. After the first step the model has one clear option with higher probability.

Initial status	% open	% closed	% holding	Nr. of transitions
open (sampled)	55.7	44.1	0.1	0.73
closed (sampled)	7.6	92.3	0.1	0.28
holding (sampled)	0.0	0.1	99.9	0.02
open (train)	53.6	46.4	0.0	1.66
closed (train)	8.4	91.6	0.0	0.36
holding (train)	0.0	0.0	100.0	0.0
open (test)	61.2	38.8	0	1.65
closed (test)	4.2	95.8	0.0	0.23
holding (test)	0.0	0.0	100.0	0.0

Table 5-1: Percentage of open, closed and holding gripper for sampled trajectories with different initial gripper status. The numbers are averaged over 100 sampled trajectories of length 25 for the top 3 rows. The numbers can be compared with the same statistics for the trajectories in the training and test set. The sampled trajectories show more or less the same statistics as the training data for the percentage of open, closed and holding. The sampled trajectories are a bit conservative when it comes to changing the gripper status. There are less transitions when starting with an open gripper than for the training data.

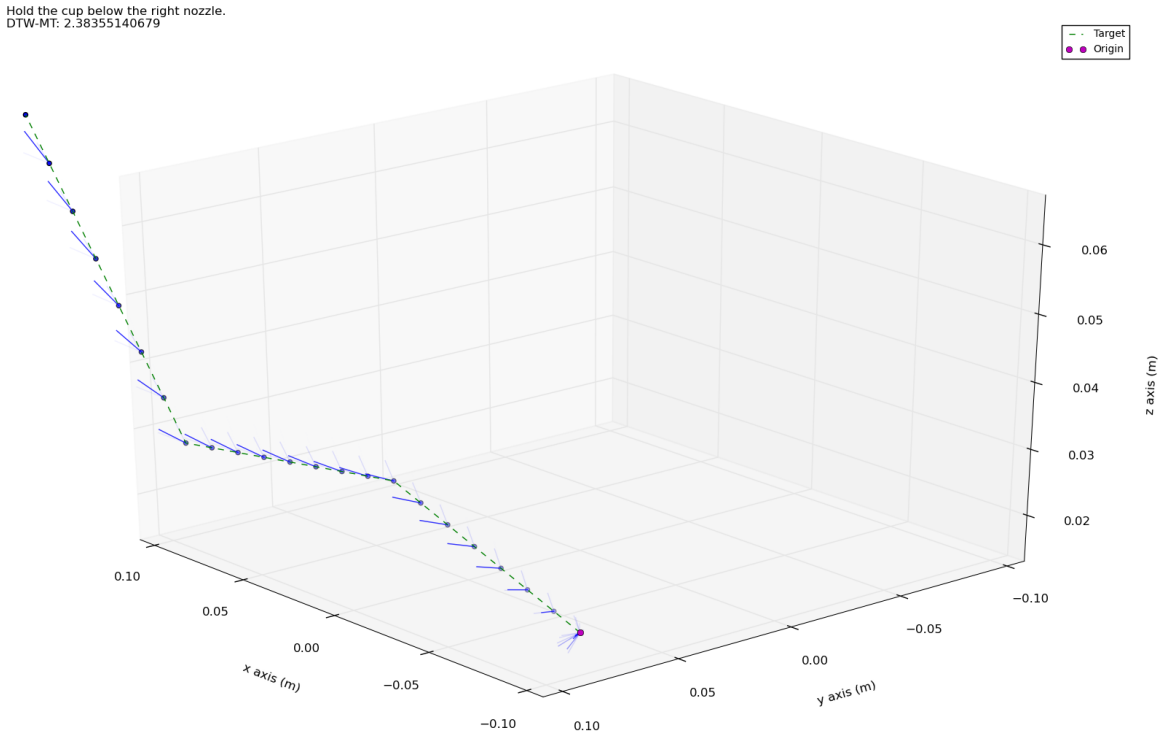


Figure 5-11: Typical example of a 'hold below' trajectory to hold a cup below a nozzle or spout. First the model has a tendency to move along the x-axis. After a while, the model wants to move more in an upwards direction.

Initial status	Total translation (m)	Total rotation (rad)
open (sampled)	0.193	0.971
closed (sampled)	0.156	0.664
holding (sampled)	0.198	0.415
open (train)	0.251	1.42
closed (train)	0.200	0.72
holding (train)	0.157	0.223
open (test)	0.265	2.65
closed (test)	0.223	2.36
holding (test)	0.150	2.16

Table 5-2: Average total translation and rotation per trajectory for 100 sampled trajectories and the complete training and test set. The sampled trajectory tend to move and rotate a bit less then the training trajectories, when starting with an open or closed gripper. For holding trajectories the sampled ones move and rotate slightly more.

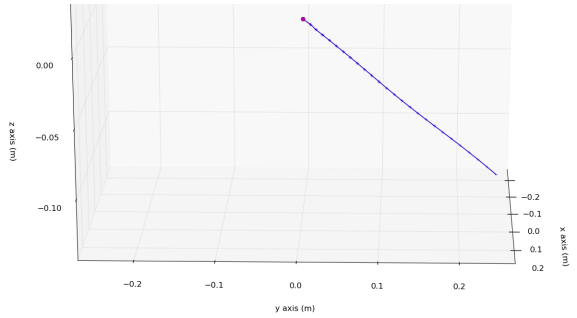
Statistics about the gripper status for the sampled trajectories can be found in Table 5-1. These numbers can be compared with the numbers for the training and test set. We see similar patterns in the sampled trajectories as for the training and test data. The number of transitions for holding patterns is close to zero. Trajectories with this gripper status do not change the gripper. Most transitions take place when the gripper starts open, then approaches an object and then closes to grasp the object. After doing some operation, like rotating, the object part is released again — the gripper is opened. The explains the higher number of transitions when starting with an open gripper status.

In Table 5-2 we look to the total translation and rotation for the sampled trajectories and compare this with the same statistics for the training and test data. We see the model has learned to produce translations and rotations of reasonable length. The numbers are about equal. The average translation is a bit less for the sampled trajectories than for the training and test one, except the holding ones. The same goes for the rotation.

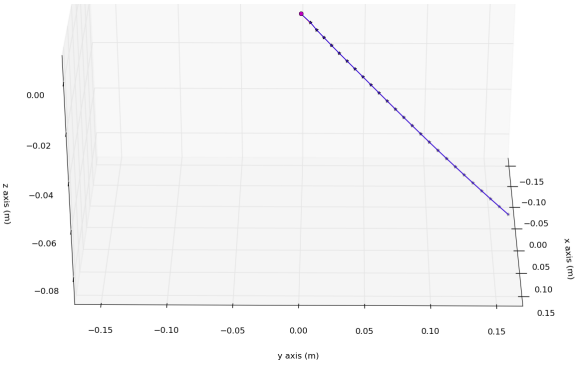
We also visualize some of the sampled trajectories. First we visualize three examples — starting with an open, closed and holding gripper — with a high bias b of 1000. The high bias b eliminates the variability in the sampling and shows the learned preferences of the model. The results are shown in Figure 5-12a, 5-12b and 5-12c. We see the holding trajectory has a smooth curve to move the gripper under a nozzle or spout. This generated trajectory looks realistic. The open and close one keep the gripper status constant and keep choosing the most save option, so they move in a straight line forward and down. This is a good start for a trajectory, however at a certain point — when doing a 'push' operation for example — the gripper has to move back. The model has not enough confidence to really do this and keeps moving forward.

By performing unbiased generation — b is set to 0 — not always the most straightforward option is chosen and more patterns in the hallucination become visible. For the sampled trajectories starting with an open gripper, we picked two trajectories where the open-close-open (grasp-release) pattern is visible (see Figure 5-13).

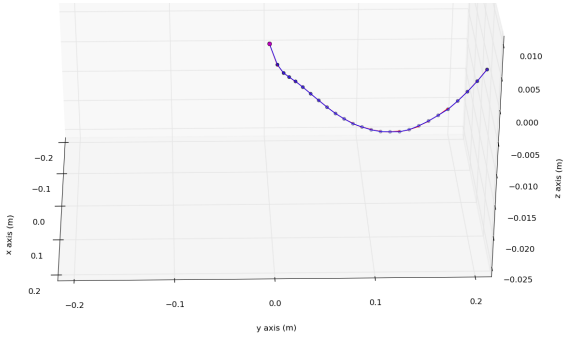
We also let the network hallucinate about trajectories with a closed gripper. This are often 'push down' or 'press' trajectories. Two hallucinated trajectories are visible in Figure 5-14.



(a) Starting with an open gripper, the sampled trajectory keeps the open gripper and moves in a straight line.



(b) Starting with a closed gripper, the sampled trajectory keeps the gripper closed and moves in a straight line.



(c) Starting with a holding gripper, the sampled trajectory moves with a smooth curve to a certain point.

Figure 5-12

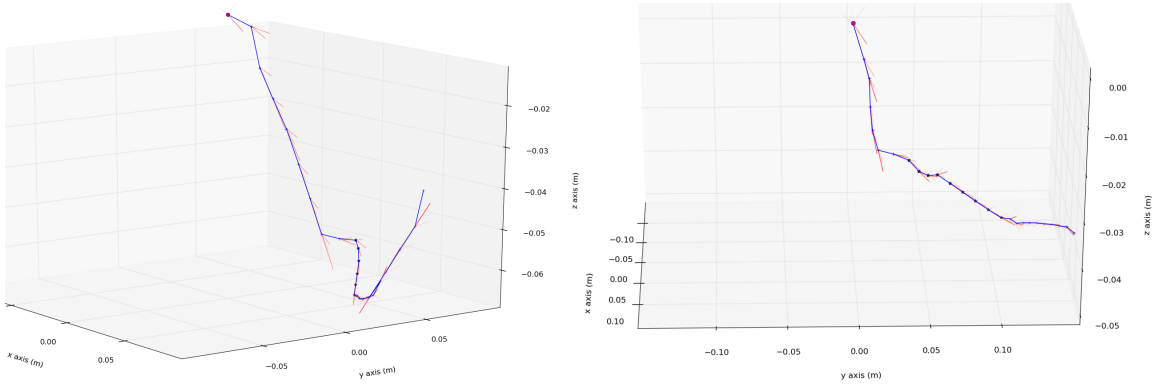


Figure 5-13: Two hallucinated trajectories starting with an open gripper. We see the open-close-open pattern (grasp-release) for both trajectories. This is a pattern that frequently occurs in the training trajectories with open gripper. The left trajectory moves in a certain direction and closes the gripper. The gripper does not moves back up, but still in a forward direction. When this was backward the trajectory would look like more realistic. Also for the right trajectory we see the open-close-open pattern, however this one keeps moving in a down and forward direction.

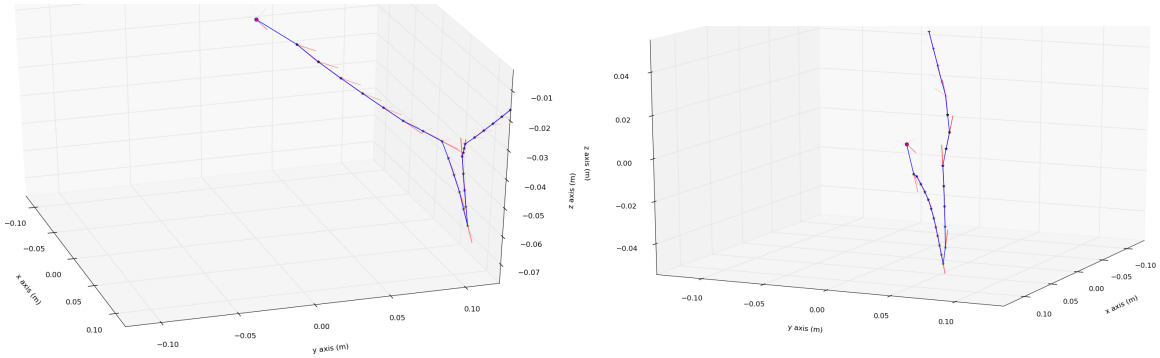


Figure 5-14: Two hallucinated trajectories starting with closed gripper. The left one can be considered as a hallucinated 'push' trajectory. The gripper stays closed, approaches something, does a 'push movement', moves in the opposite direction and moves away. Only the moving away part is not in a direction we would expect. The right one is a hallucinated 'push down' trajectory. After moving down the gripper moves back up, this pattern is clearly visible in the training trajectories. We see the last waypoints have a open gripper. This seems not reasonable. Probably this is generated, because the model does not have a clear idea when to end a trajectory.

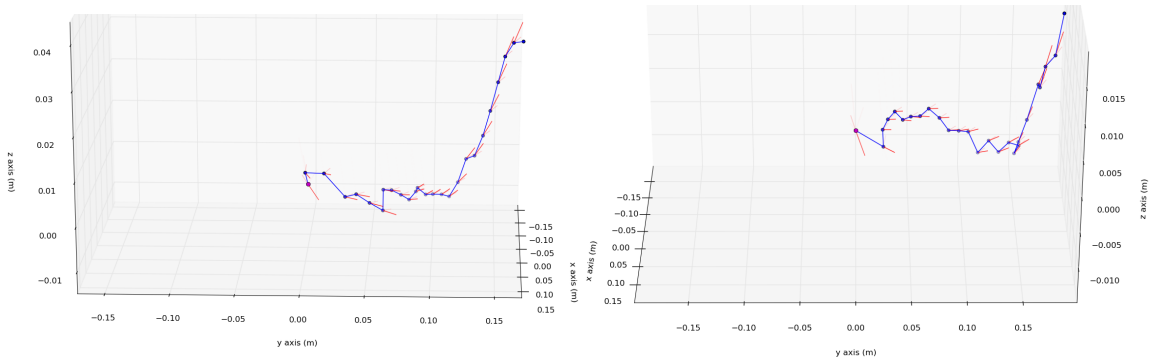


Figure 5-15: Two hallucinated holding trajectories. The trajectories are unbiased and that is why they are not very smooth. Nevertheless the trajectories move with a curve in an up and forward direction.

The same is done for holding trajectories. The results are shown in Figure 5-15. Also here the trajectories move with a holding gripper in a forward and up direction with a curve. However, because the sampling is unbiased, the trajectory is not smooth. To make smoother holding trajectories the bias b can be increased.

5-5 Trajectory Synthesis

In [3] a feature representation is learned for point-cloud/natural language instruction combinations. The goal during training is to map pairs that have similar manipulation trajectories to similar regions in the feature space. To accomplish this a special loss function is used, see Chapter 4. We want to see if we can condition the generation of trajectories on these representations.

5-5-1 Context Vectors

First we inspect how the different feature representations look like and what is learned. This is done by using a t-SNE visualization [32]. The t-SNE algorithm gives each multi-dimensional datapoint a location in a two-dimensional map. The pointcloud/language pair representations are 25-dimensional datapoints, a subset of these is used for visualization. The t-SNE algorithm is run until converge with a perplexity of 5 and a learning rate of 10. The resulting two-dimensional embedding is shown in Figure 5-16.

When we look at the corresponding language instructions for all the pairs at the top-right cluster we see very similar instructions, all with the “hold below” phrase. We mention a few examples:

- Hold the cup below the nozzle.
- Hold the cup below the white grape juice nozzle.
- Hold the cup below the spout.

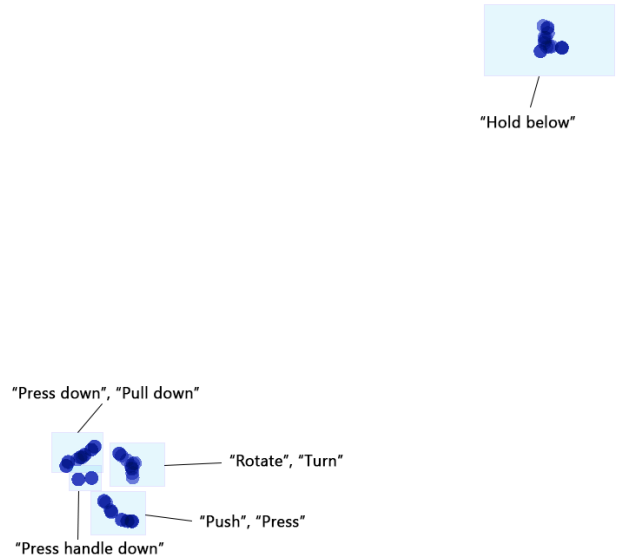


Figure 5-16: A t-SNE embedding for a subset of the pointcloud/language pairs. We clearly see two clusters, one at the top-right and one at the bottom-left. When zooming in onto the bottom-left cluster we can identify 4 subclusters.

- Hold the cup below the right nozzle.
- Hold the cup below the passion orange guava juice nozzle.

We can separate the datapoints at the bottom left in four subclusters again. Also here we can see some correspondences between the subclusters, although they are more detailed.

The subcluster at the bottom contains mostly instructions to push or press a button. A few examples:

- Push the white grape juice button.
- Press the button to open the door.
- Squeeze the trigger to make the car go forward.
- Press the button to make the water flow.
- In an emergency, hit the push-button in the back.

The subcluster at the top-left contains a lot of instructions to push down a lever or pull a handle. A number of examples:

- Press down on the stapler's handle.
- Push down on the right lever to dispense cold water.
- Press down on the handle to punch holes.
- Pull down the handle towards you to dispense the tea.

- Press down on the handle to dispense coffee.

The subcluster at the top-right contains a lot of rotate or turning in a clockwise or counter-clockwise manner instructions. A few examples:

- Rotate the handle clockwise to start the water.
- Turn the handle counterclockwise and pull to open the cabinet door.
- Rotate the knob clockwise to turn on the power.
- Rotate the knob clockwise to the desired setting.
- Turn the handle clockwise to fill the cup.

The subcluster at the middle-left contains instructions to press down a handle

- Press down on the stapler's handle.
- Press down on the handle to punch holes.
- Press down on the flush handle to flush.

When we condition the trajectory generation on the feature representations of the point-cloud/language pairs we expect to see different trajectories for the "hold below" cluster than for the trajectories in the other part of the t-SNE visualization. The generated "hold below" trajectories should have a holding gripper status.

5-5-2 Training

We train the model with the as extra input the context $\mathbf{c}$ — augmented with the starting position and orientation — in the same way as the model for unconditioned generation, using the Adam algorithm for 100 epochs. The loss is shown in Figure 5-17. We see the loss is decreasing over time.

We also show performance on the RMSE — the average root mean squared eror (RMSE) between the target and predicted translation — in Figure 5-18. The performance on the DTW-MT metric is reported in Figure 5-19

5-5-3 Results

We are interested in the quality of the trajectories synthesized with the RNN model. Therefore the RNN model is used to generate trajectories for pointcloud/language pairs it has never seen before. The pointcloud and natural language instruction for this new object are encoded into the context vector $\mathbf{c}$. With this context vector $\mathbf{c}$, augmented with the starting pose (position and orientation) as input, our model generates a trajectory of 25 waypoints. Our test set consists of 60 pointcloud/language pairs.

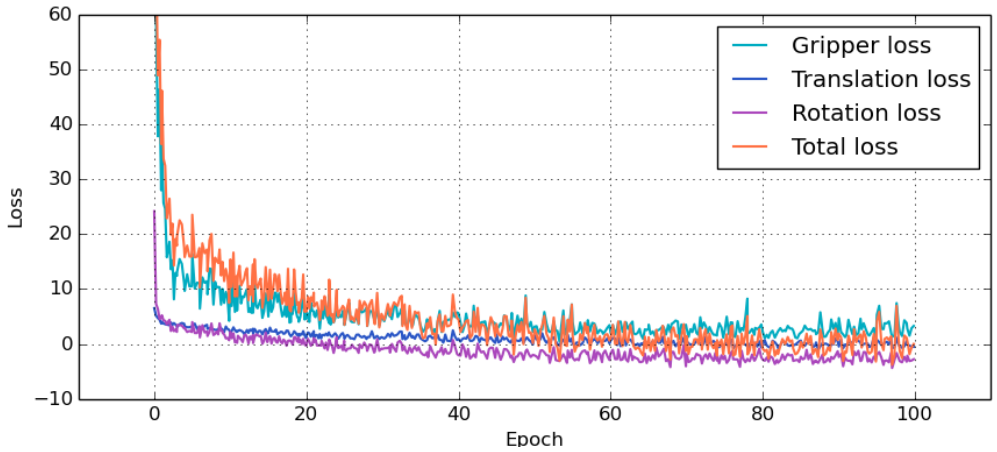


Figure 5-17: The loss is decreasing over time. We show the individual contributions of the gripper, translation and rotation term to the loss function. The gripper loss is weighted by a factor of 100, which makes this contribution the biggest during the initial phases of training. All three contributions decrease over time.

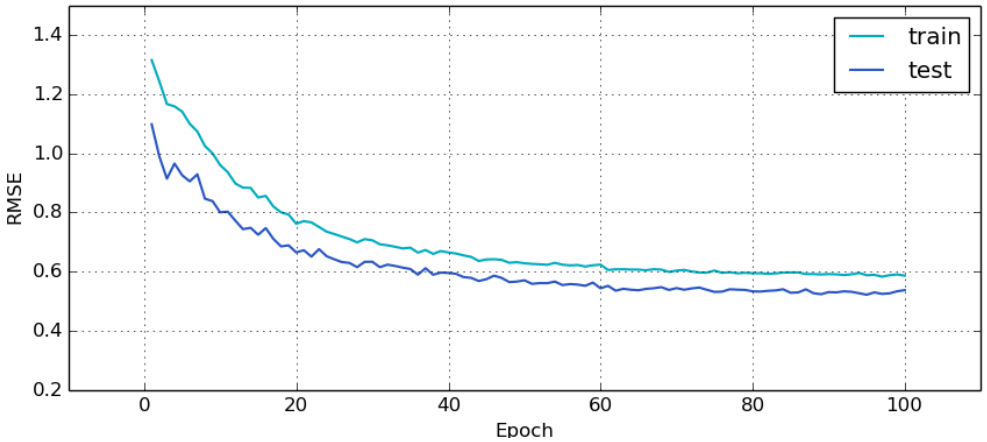


Figure 5-18: The root mean squared error for both the training and test set during training for the synthesis model. Both values are decreasing over time. The test set performs slightly better. A possible explanation for this uncommon phenomena is that the test set performs already better when the training has not started yet.

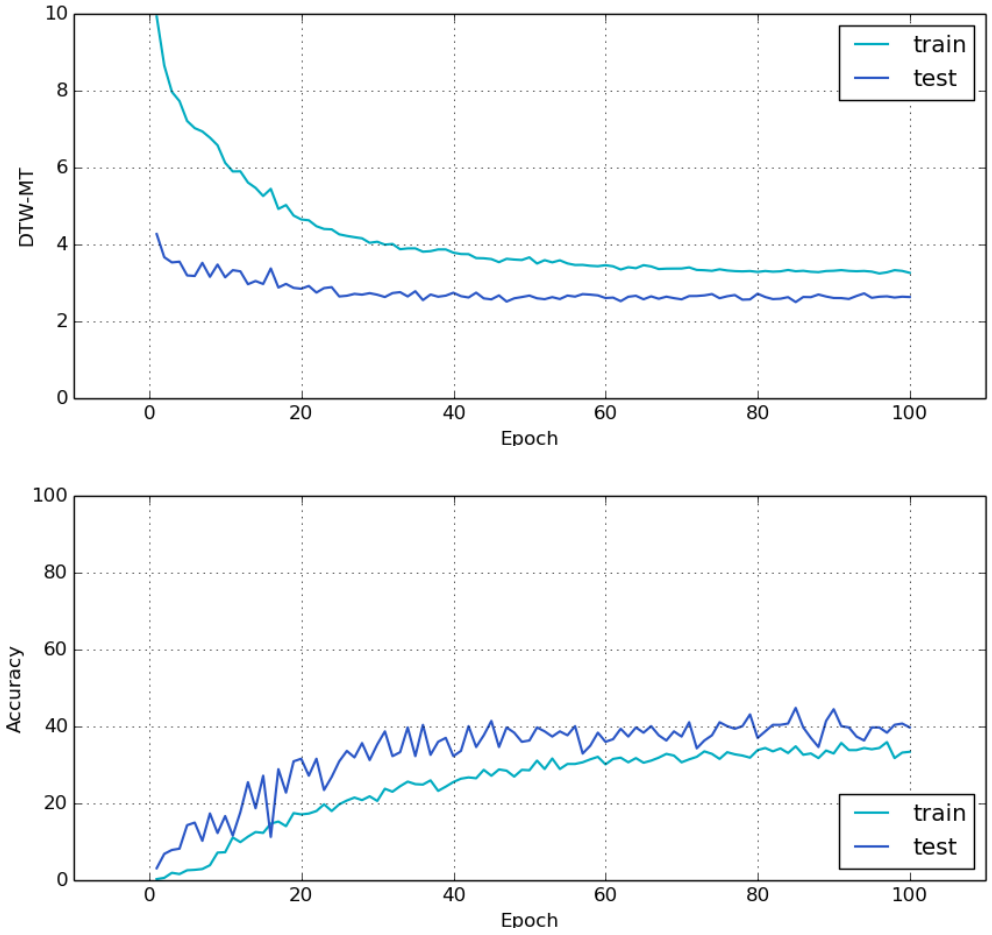


Figure 5-19: The DTW-MT for the training and test set during training and the accuracy (DTW-MT < 2).

Model	DTW-MT	Accu. (%)
Our RNN model (no context)	10.7	50.0
Our RNN model (starting pose)	7.5	62.6
Our RNN model (context without starting pose)	7.6	65.3
Our RNN model	**6.8**	**76.7**

Table 5-3: Results on the Robobarista Dataset for our RNN model. We compare our synthesis RNN model with our RNN model without context. The model without context hallucinates a trajectory. Each column shows a different metric used to evaluate the models. For the DTW-MT metric, lower values are better. For Accuracy (DTW-MT < 10), higher is better. We also include two baselines, one with only the starting pose in the context vector c and one with the context vector c without the starting pose.

For evaluation purposes an expert trajectory, which is not included in the training data, is used. Each generated trajectory is evaluated against this expert demonstration, using the DTW-MT measure. We choose to let the generated trajectory start at the same position as the expert trajectory, with the same orientation.

Then we use the DTW-MT metric to measure the loss between the trajectories. With this metric all generated waypoints are matched against waypoints in the expert trajectory (or the other way around if the generated trajectory has more waypoints than the expert trajectory). The starting points are matched against each other and the same holds for the end points. The intermediate waypoints are matched against other waypoints in the expert trajectory. Order is preserved, this means the third waypoint of the generated trajectory cannot be matched against the second expert waypoint, when the second generated waypoint is already matched against the third expert waypoint.

We compare our RNN method with some baselines, the results are in Table 5-3. We see that providing the context performs significantly better than a hallucinated trajectory from Section 5-4-2. Because the hallucinated trajectory starts at the same point as the expert trajectory the computed loss on the first few waypoints is relatively low. Therefore this baseline model has still a reasonable DTW-MT score and accuracy, although it does not know what kind of task to perform. We also analyse the effect of adding the starting pose. Adding only the starting pose — the other entries in the context c are set to zero — performs better than with no context at all. The same holds for a context without starting pose — these entries are set to zero. So from the numbers in Table 5-3 we can see that both the starting pose and the context (the learned pointcloud/language embedding) help to generate trajectories closer to the expert ones.

We also compare our method to the current state-of-the art, the method that finds the nearest trajectory in the learned multimodal embedding space [3]. We call this method nearest neighbour. We also include a baseline from [3] called chance, this method picks a random trajectory from the set of training trajectories.

To make a fair comparison with the nearest neighbour method, we let our RNN model start at the same position (and with same orientation) as the nearest neighbour trajectory. Both the nearest neighbour trajectory and the generated trajectory are compared with the expert with the DTW-MT loss. The results are in Table 5-4. We see that the average loss of our generated trajectory is higher than for the nearest neighbour method. However, we note that the nearest neighbour method does not have the flexibility to start at different positions.

Model	DTW-MT	Accu. (%)
Nearest Neighbour [3]	**9.2**	**76.7**
Our RNN model (no context)	15.3	20.0
Our RNN model	11.4	55.0

Table 5-4: Scores when our RNN model starts at the same point as Nearest Neighbour. Also a hallucinated trajectory for each pointcloud/language pair is generated as a baseline.)

Model	Open	Close	Hold
Chance [3]	0.49	0.65	0.49
Nearest Neighbour [3]	**0.17**	**0.15**	0.26
Our RNN model (no context)	0.32	0.55	0.48
Our RNN model	0.18	0.27	**0.16**

Table 5-5: Results on Robobarista Dataset. Rows list models we tested, including our RNN model and baselines. The percentage of open, close and hold gripper statuses per trajectory is computed. The absolute difference with respect to the expert trajectory is averaged over all pointcloud/language pairs in the test set. Lower numbers are better.

To further analyse our generated trajectories we look to the gripper status only. Per trajectory we compute the percentage of open, close and holding. The same is done for the expert trajectories. The absolute difference per gripper status is computed. The average numbers for the test set are reported in Table 5-5.

We see our RNN model performs better than the chance model and the RNN model without context. The nearest neighbour method performs best. It turns out our RNN model does almost always predict the right waypoint for a task where it has to hold a cup below a spout or nozzle. That this kind of task is relatively easy to identify from the context vector c is clear from the t-SNE embedding shown in Figure 5-16. The number for the holding status for our RNN model is still 0.16, this is due to the fact that the expert does also some 'push' tasks with a holding gripper.

We show some examples of generated trajectories in the Figures ranging from 5-20 to 5-25. It is clear that for most 'Hold below' tasks the generated trajectory looks reasonable and has for a number of examples a lower DTW-MT loss than the nearest neighbour trajectory. For some pretty good trajectories with low DTW-MT loss, see Figure 5-20 and 5-21. The 'Hold below' trajectories can be considered as the most easy, because here the movement is only in one direction and the gripper status is constant. Also a significant part of the training trajectories, almost half of them, consist of this kind of trajectories. Because these trajectories are relatively easy, these training trajectories look also quite similar. This makes it easier for the RNN to learn reasonable 'Hold below' trajectories. For the other kind of trajectories the variance is much higher, trajectories for a 'Pull down' instruction can be quite different and have movements in different directions. Together with the fact that we have a relative small amount of data, it turns out that it is difficult for the RNN to synthesize reasonable 'push', 'pull down', 'press' or 'rotate' trajectories.

Also the nearest neighbour method seems to have reasonable performance for 'hold below' trajectories, but more difficulties with the other one. Also this seems logical, because a 'hold below' trajectory can be easier transferred to another pointcloud/language pair. This

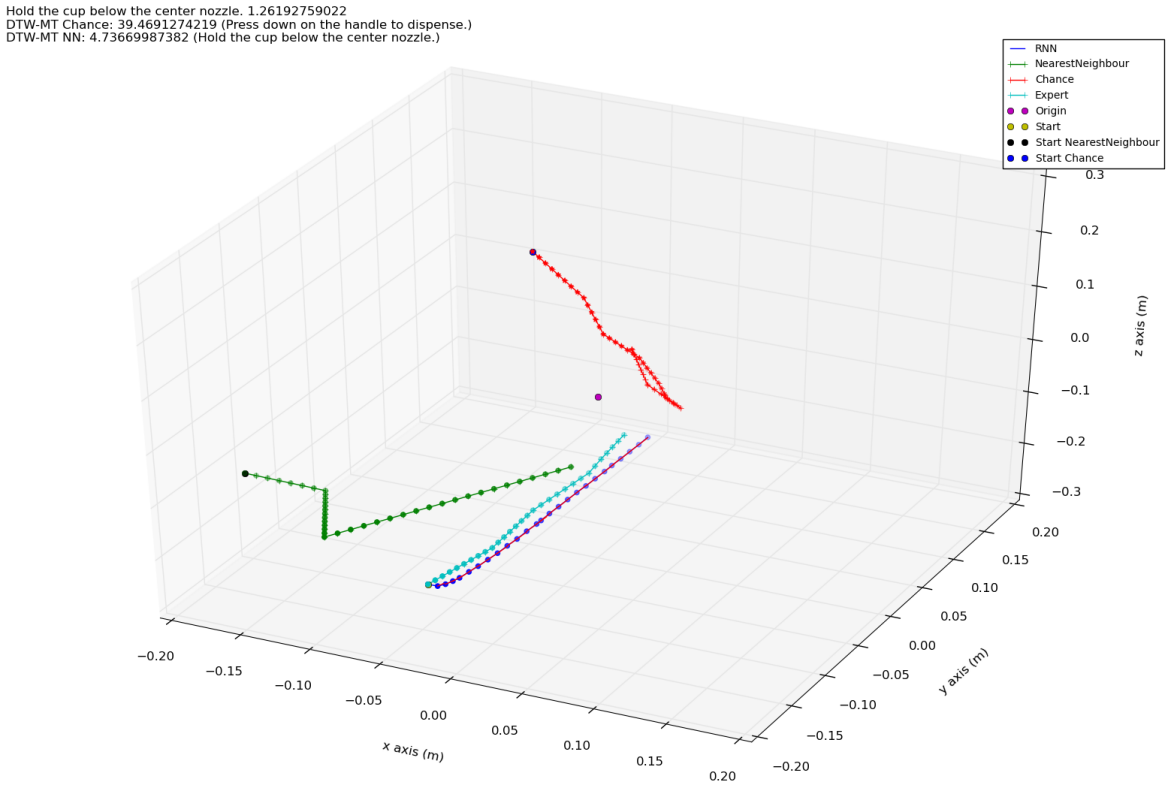


Figure 5-20: A generated 'hold below' trajectory very similar to the expert one. Also the loss (1.26, plotted after the instruction) is for this example better than for the nearest neighbour trajectory, which has a DTW-MT loss of 4.74. We also mention the instruction corresponding originally to the chance and nearest neighbour trajectory.

is because these kind of trajectories are quite similar. For other tasks this turns out to be more difficult.

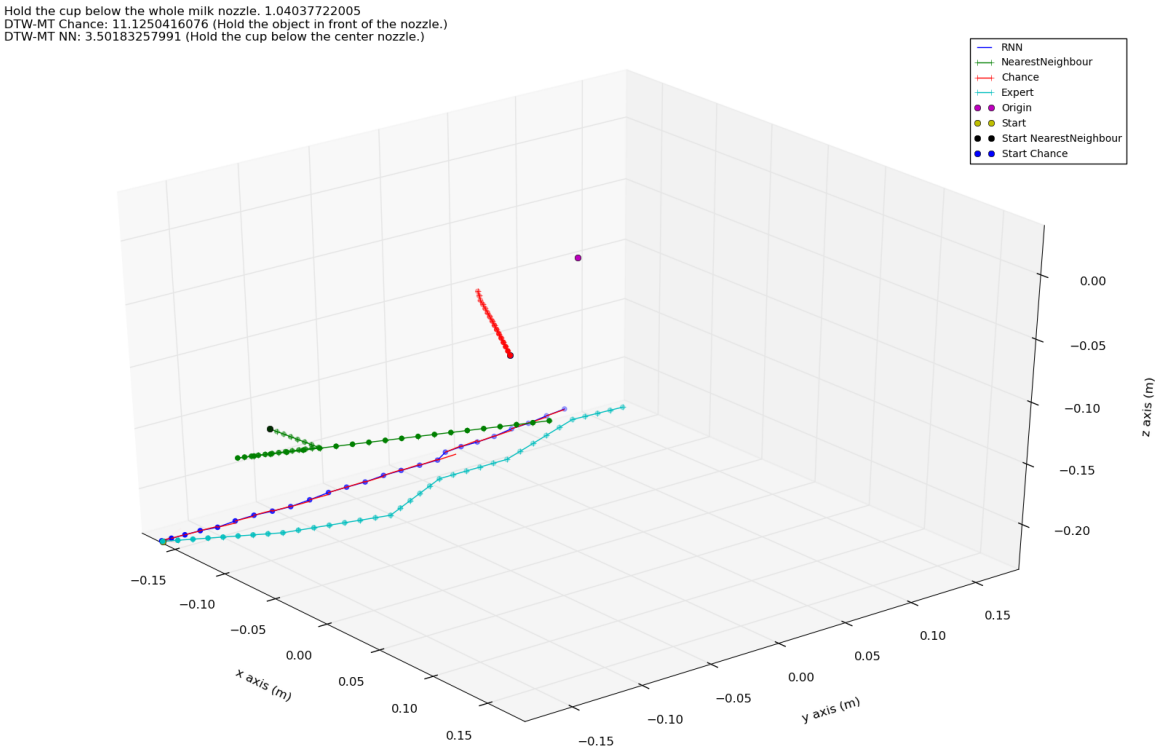


Figure 5-21: Another generated 'hold below' trajectory similar to the expert one. In red the means of the mixture components are plotted.

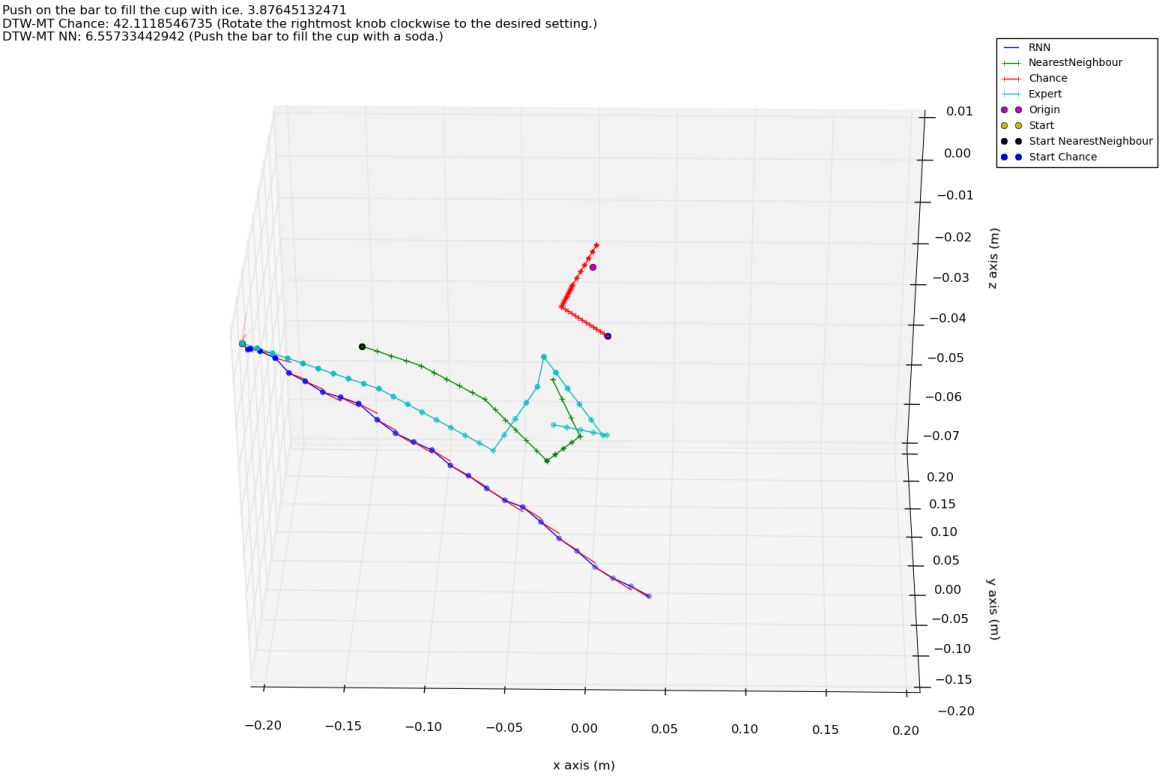


Figure 5-22: A 'push' trajectory, where the expert chooses to do this with a holding gripper. The generated trajectory initially moves in the right direction, but does not perform any 'push' and moving back movement. In red the means of the mixture components are plotted.

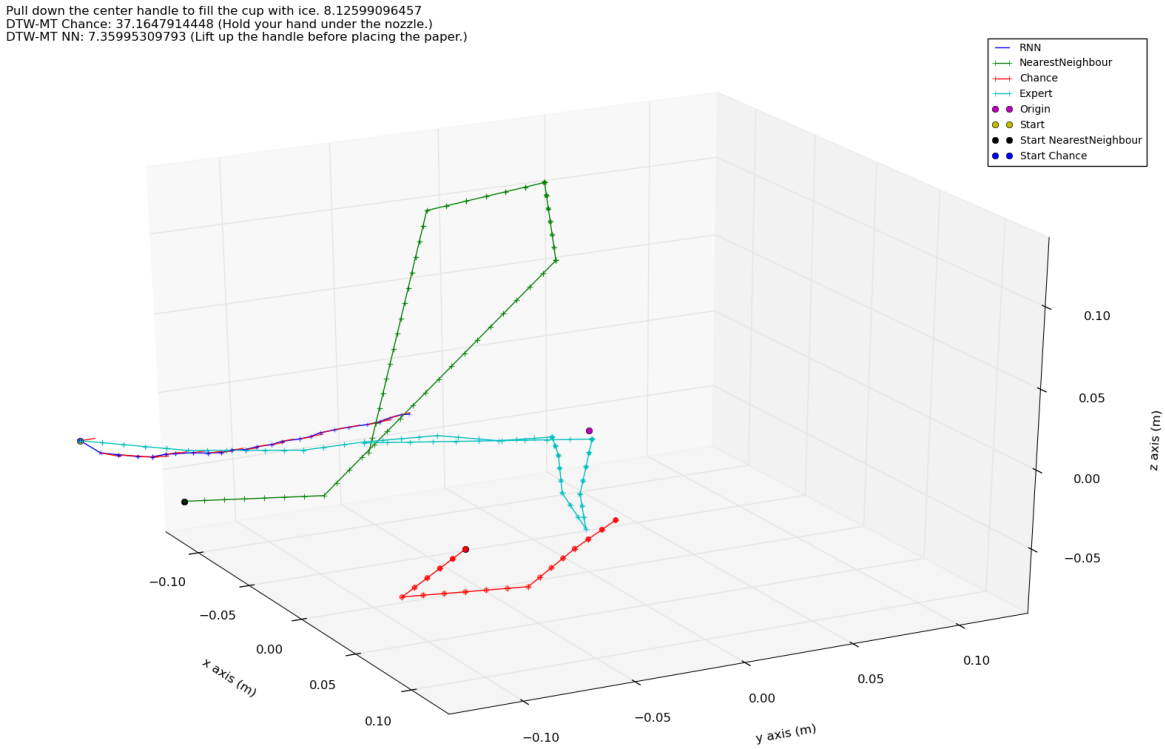


Figure 5-23: Here we see a generated trajectory for a 'pull down' task. The expert approaches the handle with an open gripper status (+), after which the gripper moves down and moves back up with a closed gripper (*). Then it moves back with an open gripper (+). The generated trajectory only starts approaching the object with an open gripper. In red the means of the mixture components are plotted.

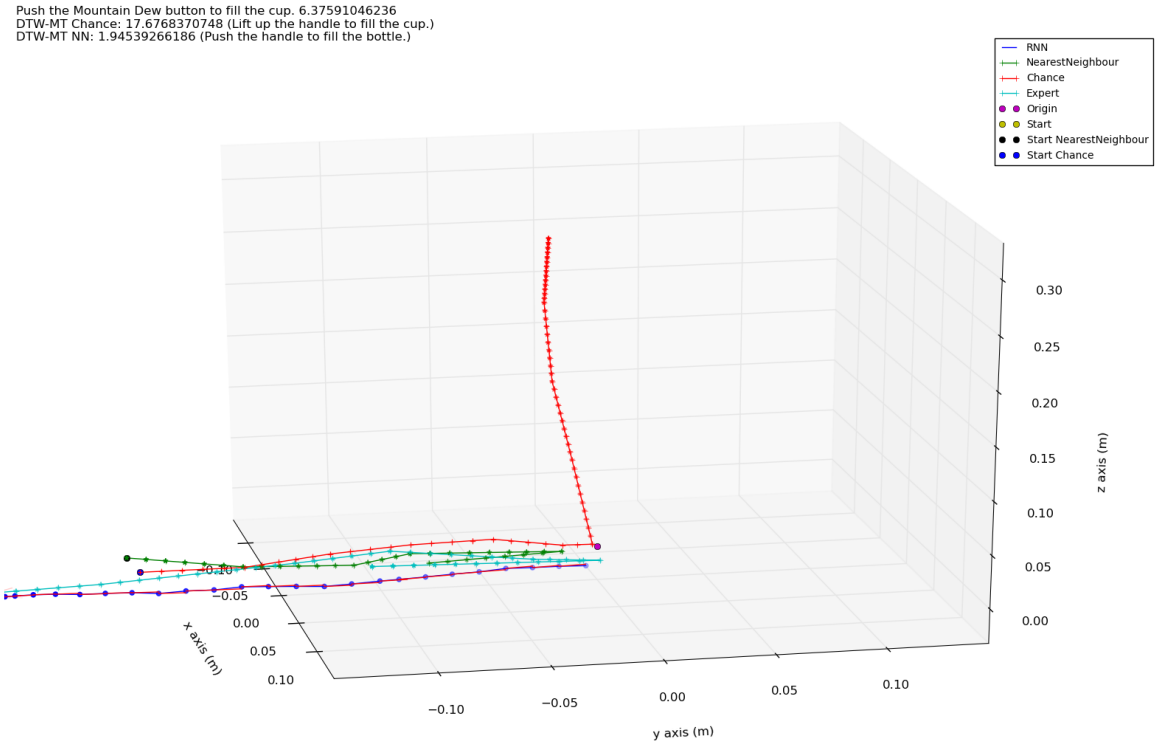


Figure 5-24: A 'push' trajectory, the expert approaches the button with a closed gripper and moves back. The generated trajectory moves in the right direction, however with a closed gripper and it does not move back. The nearest neighbour trajectory is very close to the expert for this example.

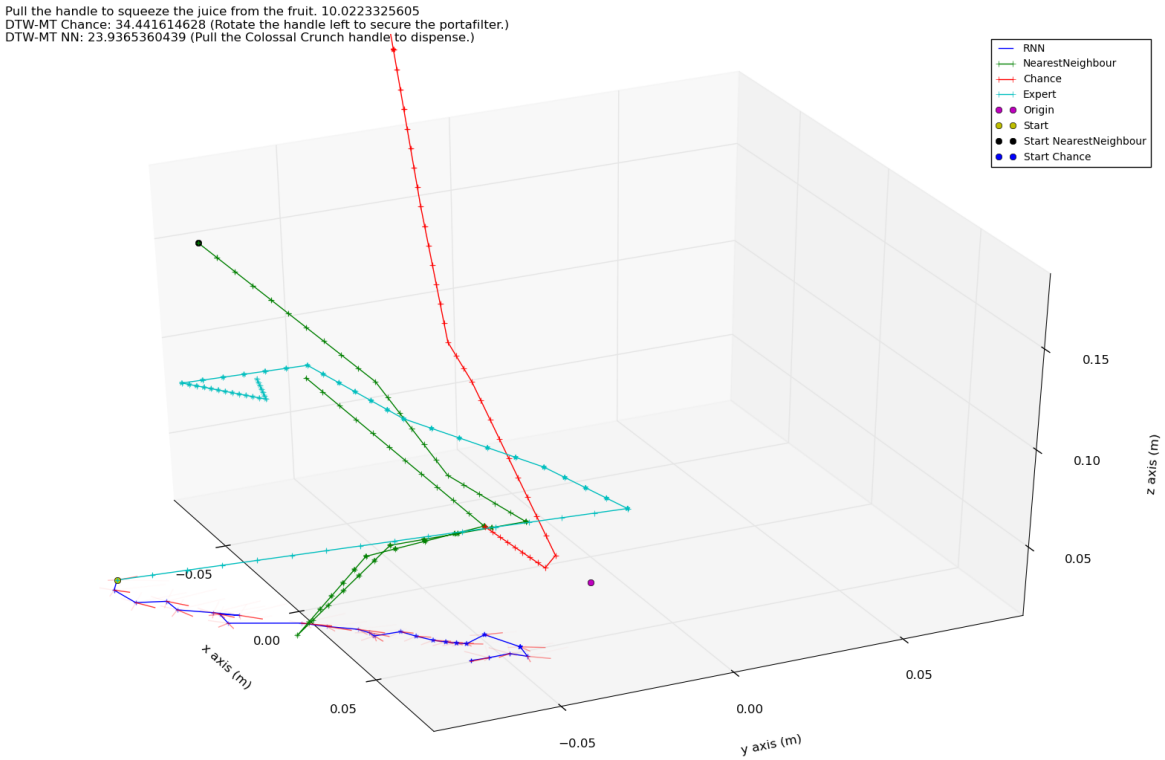


Figure 5-25: Another example for an example that is difficult. The generated trajectory starts with an open gripper, same as for the expert. The generated trajectory closes the gripper and starts to move in another direction. In red the means of the mixture components are plotted.

Chapter 6

Conclusions and Future Work

In this work, we approached the task of generating robotic manipulation trajectories for common tasks in a household environment. We proposed a *learning from demonstrations* strategy. With this strategy a recurrent neural network is trained to predict the next waypoint of the manipulation trajectory. By iteratively sampling from this predictive distribution complete trajectories could be generated. This approach has successful applications in other domains, including text, handwriting synthesis, audio and image caption generation.

First we approached the generation of trajectories, without any context involved, also called 'hallucinating'. The recurrent neural network has to learn to exploit patterns in the waypoints seen so far, to predict the next waypoint. We showed that — by sampling trajectories from our trained model — the network has succesfully learned reasonable patterns. These patterns become most clearly visible in 'hold below' trajectories, but also patterns for 'push', 'pull down' or 'press' actions emerged. This showed the capability of generating reasonable trajectories.

Second we extended our unconditioned model to a conditioned case, trajectory synthesis. Here the trajectory generation is conditioned on the task in natural language and the object part as a point cloud. Work from [3] is used to encode this multimodal input into a compact context vector. In [3] the nearest trajectory from the complete training set in the learned feature space is selected. This limits the flexibility when a manipulation trajectory is required for a new unseen environment, because the existing trajectory is used as it is. It is not adjusted to the small differences between environments and the same starting position has to be used. In our method the generation of the trajectory is then conditioned on the learned compact representation of the object part and task in natural language, so here we showed an encoder-decoder structure with a mapping from pointcloud/language pairs to trajectories. We trained and tested our algorithm on the Robobarista dataset [19]. We showed that our approach does generate reasonable trajectories for simple tasks ('hold cup below'). However, we showed that for difficult and challenging tasks our approach does not generate reasonable trajectories. Some reasonable patterns emerged for some examples, such as a right gripper status. Anyhow, we showed that this is not enough to perform the instruction successfully.

To generate better trajectories for the challenging tasks, it is likely that a lot more training data is needed.

6-1 Future Work

For future work we have a number of ideas and suggestions:

- The main bottleneck with our current approach for trajectory synthesis is the need for lots of training data. After training we see reasonable patterns emerging, but not yet enough for complete reasonable trajectories. Therefore we expect that our method will benefit from methods as one-shot learning [40] or other methods that require less data. A method called active one shot learning is described in [41]. When this method is applied to our application of generating trajectories, the trajectories are presented in a stream during training. Then at each step a decision has to be made whether to predict the next label (the correct trajectory) or pay to receive the correct label. The optimal strategy would be to maintain a set of pointcloud/language pair representations and their corresponding trajectory representations, in memory. Then, upon receiving a new pointcloud/language pair, the optimal strategy is to compare this to the existing pointcloud/language representations in memory. Here the uncertainty of a match has to be weighted against the cost of requesting a label. If the model believes it is a new pointcloud/language pair — not comparable to the memorized representations — then a representation for this pair must be stored, the label requested, and stored and associated with the new representation.

- In the current approach the training of the encoder (pointcloud/lanugage pairs to context vector) and the decoder (context vector to next waypoint) is completely separated. This could be combined to train a fully end-to-end network. This means the error when predicting the next waypoint during training is also backpropagated to the weights of the encoder network. This could lead to context vectors better suited to generate trajectories from. Another possibility is to also train the context vectors itself during the trajectory synthesis training.

- Note that the current approach for generating trajectories — sampling one waypoint per time step — is a greedy approach. Only one waypoint per step is sampled and used as the next input. A more advanced option is to use a beam search heuristic. Beam search is a heuristic search algorithm that explores a graph by expanding the most promising nodes in a limited set. For generating trajectories this means that at each step a number of options with high probability for the next waypoint are considered and kept as partial hypotheses. At each step then a number of these partial trajectories are further expanded until a number of trajectories is complete. Then some criteria has to be used to select the final trajectory from the set. More decoding strategies, with good results in machine translations, can be found in [42].

- To improve the learning during training the concept of incremental sequence learning [12] can be useful. The network must learn the ability to build up internal representations of the part of the sequence received so far. This is necessary to predict the next step of the sequence. The key observation is: “later steps in the sequence can only

be learned well once the network has learned to develop the appropriate internal state summarizing the part of the sequence seen so far". Therefore it may be beneficial to learn first the very first steps in the sequence. When an suitable internal representation of this part of the sequence has been developed the length of the sequences is gradually increased. With using this method an improvement in sequence learning performance is found in [12].

- With the method described in this chapter we predict iteratively the next item of the sequence. This can lead to a problem if the wrong decision is taken at a certain step $t-1$. In that case, the model can become in a part of the state space that is very different from those visited during training. The network will not know what to do and this can lead to cumulative bad predictions. To deal better with this case, [43] proposes to use during training at step t the true step to predict the next step *or* the one that is predicted during the previous step. In this way the model learns to deal with small errors in the predictions. So during training randomly is decided to use the true current step or the predicted estimate. In the first rounds, when the model is not well trained, higher probability is given to the true step. During training, when the model gets better, the probability of using the predicted one becomes higher.

- It will be interesting to really use variable length sequences instead of sequences interpolated to a fixed length. When using variable length sequences, an end-of-sequence marker has to be used as extra input and output. The network has then to predict when to end a sequence. This can improve the trajectories, because it now sometimes looks like something new is started, when the trajectory has already done something.

- One issue we have is that better performance on the metrics for predicting the next waypoint do not necessarily lead to better generated trajectories. It would be interesting to train a discriminator network that learns to identify between human generated trajectories and trajectories generated with our method. Feedback of this method can be used in some way to augment the loss function during training. This is the same principle as used for generating naturally looking images with Generative Auto Encoders (GANs) [28].

- We used the encoding for the pointcloud/language pairs from [3]. It is interesting to research which information is really encoded in the learned context vectors and whether there is some structure in the learned embedding space. When learning word vectors it was found that `vector("King") - vector("Man") + vector("Woman")` results in a vector that is closest to the vector representation of the word Queen. It is worth researching whether this is also possible with learned representations for natural language instructions and object parts.

Appendix A

Quaternions

In this work quaternions are used to represent orientations and rotations: orientations with respect to a fixed coordinate frame and rotations between consecutive steps.

We need a representation that represents the orientation of the gripper of the robot arm. We have two strong preferences for this representation: (i) two orientations that are physically close should be close in the representation and (ii) the representation should be unique. All orientations that look identical should have the same value. The problem of finding a correct way to represent a pose is discussed in more detail in [44].

We choose to use the same representation as in [3], *quaternions*. A quaternion $\mathbf{q} \in \mathbb{R}^4, ||\mathbf{q}|| = 1$ can be used to represent an orientation in a 3-dimensional environment. A unit quaternion represents a rotation by an angle θ around a unit axis vector $\mathbf{a}$ as:

$$\mathbf{q} = \begin{bmatrix} a_x \sin\frac{\theta}{2} & a_y \sin\frac{\theta}{2} & a_z \sin\frac{\theta}{2} & \cos\frac{\theta}{2} \end{bmatrix} \tag{A-1}$$

A quaternion is often the preferred choice over Euler angles (yaw, roll and pitch). Euler angles can be ambiguous (see Figure A-1), a key reason for this is that a phenomena like Gimbal lock[1] can occur. We can think of a quaternion as 3 numbers representing an axis and the fourth number the amount of rotation around that axis. However, also quaternions are not completely unambiguous, q and $-q$ represent exactly the same orientation.

Four output values at the output layer of a neural network are easily mapped to legitimate rotations by normalizing them to unit length, this method is used in [37] as well. In [37] a neural network is used to estimate the camera pose from a single image. A quaternion is here preferred over a rotation matrix. This is because normalizing a 4D-value to unit length is simpler than the orthonormalization required for rotation matrices.

[1] GimbalLock explained: https://www.youtube.com/watch?v=zc8b2Jo7mno

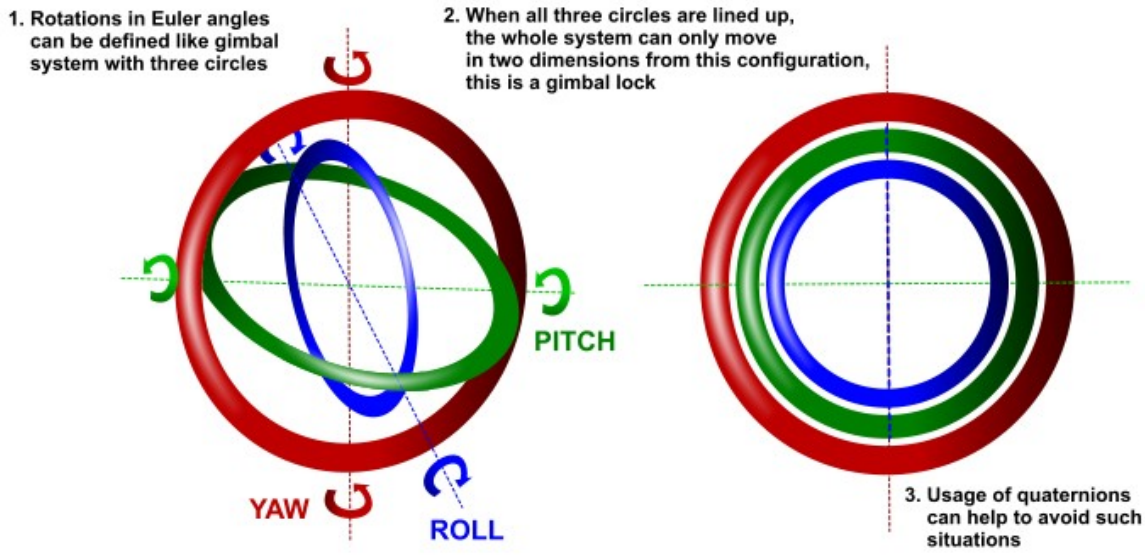


Figure A-1: The reason why quaternions are often preferred over Euler angles. Image from http://3dvrm.com/quat/.

Bibliography

[1] A. Karpathy, "Neural Networks Part 1: Setting up the Architecture." CS231n: Convolutional Neural Networks for Visual Recognition, 2015.

[2] A. Graves, "Generating sequences with recurrent neural networks," *CoRR*, vol. abs/1308.0850, 2013.

[3] J. Sung, I. Lenz, and A. Saxena, "Deep multimodal embedding: Manipulating novel objects with point-clouds, language and trajectories," in *International Conference on Robotics and Automation*, 2017.

[4] C. G. Atkeson, "DARPA Robotics Challenge: Team Steel." http://www.cs.cmu.edu/~cga/drc/.

[5] B. D. Argall, S. Chernova, M. Veloso, and B. Browning, "A survey of robot learning from demonstration," *Robotics and Autonomous Systems*, vol. 57, pp. 469–483, May 2009.

[6] P. Abbeel, M. Quigley, and A. Y. Ng, "Using inaccurate models in reinforcement learning," in *Proceedings of the 23rd International Conference on Machine Learning*, pp. 1–8, 2006.

[7] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[8] R. Józefowicz, O. Vinyals, M. Schuster, N. Shazeer, and Y. Wu, "Exploring the limits of language modeling," *CoRR*, vol. abs/1602.02410, 2016.

[9] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Advances in Neural Information Processing Systems*, pp. 3104–3112, 2014.

[10] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *CoRR*, vol. abs/1409.0473, 2014.

[11] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, J. Klingner, A. Shah, M. Johnson, X. Liu, L. Kaiser, S. Gouws,

Y. Kato, T. Kudo, H. Kazawa, K. Stevens, G. Kurian, N. Patil, W. Wang, C. Young, J. Smith, J. Riesa, A. Rudnick, O. Vinyals, G. Corrado, M. Hughes, and J. Dean, "Google's neural machine translation system: Bridging the gap between human and machine translation," *CoRR*, vol. abs/1609.08144, 2016.

[12] E. D. de Jong, "Incremental sequence learning," *CoRR*, vol. abs/1611.03068, 2016.

[13] S. Mehri, K. Kumar, I. Gulrajani, R. Kumar, S. Jain, J. Sotelo, A. Courville, and Y. Bengio, "SampleRNN: An unconditional end-to-end neural audio generation model," *CoRR*, vol. abs/1612.07837, 2016.

[14] A. van den Oord, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. W. Senior, and K. Kavukcuoglu, "Wavenet: A generative model for raw audio," *CoRR*, vol. abs/1609.03499, 2016.

[15] D. Bahdanau, J. Chorowski, D. Serdyuk, P. Brakel, and Y. Bengio, "End-to-end attention-based large vocabulary speech recognition," *CoRR*, vol. abs/1508.04395, 2015.

[16] J. Chorowski, D. Bahdanau, D. Serdyuk, K. Cho, and Y. Bengio, "Attention-based models for speech recognition," in *Advances in Neural Information Processing Systems*, pp. 577–585, 2015.

[17] K. Xu, J. Ba, R. Kiros, K. Cho, A. Courville, R. Salakhudinov, R. Zemel, and Y. Bengio, "Show, attend and tell: Neural image caption generation with visual attention," in *Proceedings of the 32nd International Conference on Machine Learning*, pp. 2048–2057, 2015.

[18] K. Cho, A. C. Courville, and Y. Bengio, "Describing multimedia content using attention-based encoder-decoder networks," *CoRR*, vol. abs/1507.01053, 2015.

[19] J. Sung, S. H. Jin, and A. Saxena, "Robobarista: Object part-based transfer of manipulation trajectories from crowd-sourcing in 3d pointclouds," in *International Symposium on Robotics Research (ISRR)*, 2015.

[20] E. Mansimov, E. Parisotto, L. J. Ba, and R. Salakhutdinov, "Generating images from captions with attention," *CoRR*, vol. abs/1511.02793, 2015.

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

[22] "Recurrent neural networks tutorial, Part 1 - Introductions to RNNs." http://www.wildml.com/2015/09/recurrent-neural-networks-tutorial-part-1-introduction-to-rnns/, 2015.

[23] R. J. Williams and D. Zipser, "Gradient-based learning algorithms for recurrent networks and their computational complexity," in *Back-propagation: Theory, Architectures and Applications*, pp. 433–486, 1995.

[24] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, pp. 1735–1780, Nov. 1997.

[25] C. Bishop, "Mixture density network," tech. rep., 1994.

[26] T. Tieleman and G. Hinton, "Lecture 6.5—RmsProp: Divide the gradient by a running average of its recent magnitude." COURSERA: Neural Networks for Machine Learning, 2012.

[27] A. Karpathy, P. Abbeel, G. Brockman, P. Chen, V. Cheung, R. Duan, I. Goodfellow, D. Kingma, J. Ho, R. Houthooft, T. Salimans, J. Schulman, I. Sutskever, and W. Zaremba, "Generative models." https://openai.com/blog/generative-models/, 2016.

[28] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," in *Advances in Neural Information Processing Systems*, pp. 2672–2680, 2014.

[29] D. P. Kingma and M. Welling, "Auto-encoding variational bayes," *CoRR*, vol. abs/1312.6114, 2013.

[30] A. Karpathy, "The unreasonable effectiveness of recurrent neural networks." http://karpathy.github.io/2015/05/21/rnn-effectiveness/, 2015.

[31] A. van den Oord, N. Kalchbrenner, and K. Kavukcuoglu, "Pixel recurrent neural networks," in *Proceedings of the 33nd International Conference on Machine Learning*, pp. 1747–1756, 2016.

[32] L. van der Maaten and G. E. Hinton, "Visualizing high-dimensional data using t-sne," *Journal of Machine Learning Research*, vol. 9, pp. 2579–2605, 2008.

[33] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *CoRR*, vol. abs/1412.6980, 2014.

[34] S. Ioffe, "Batch renormalization: Towards reducing minibatch dependence in batch-normalized models," *CoRR*, vol. abs/1702.03275, 2017.

[35] K. Hsiao, S. Chitta, M. Ciocarlie, and E. G. Jones, "Contact-reactive grasping of objects with partial shape information," in *International Conference on Intelligent Robots and Systems*, 2010.

[36] A. Karpathy, "ConvNet Tips and Tricks: squeezing out the last few percent." CS231n: Convolutional Neural Networks for Visual Recognition, 2015.

[37] A. Kendall, M. Grimes, and R. Cipolla, "Convolutional networks for real-time 6-dof camera relocalization," *CoRR*, vol. abs/1505.07427, 2015.

[38] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *Journal of Machine Learning Research*, vol. 15, pp. 1929–1958, 2014.

[39] D. Ha, "Generative handwriting demo using tensorflow." https://github.com/hardmaru/write-rnn-tensorflow. Accessed: 2017-01-09.

[40] A. Santoro, S. Bartunov, M. Botvinick, D. Wierstra, and T. P. Lillicrap, "One-shot learning with memory-augmented neural networks," *CoRR*, vol. abs/1605.06065, 2016.

[41] M. Woodward and C. Finn, "Active one-shot learning," in *Advances in Neural Information Processing Systems*, pp. 1747–1756, 2016.

[42] J. Gu, K. Cho, and V. O. Li, "Trainable greedy decoding for neural machine translation," *CoRR*, vol. abs/1702.02429, 2017.

[43] S. Bengio, O. Vinyals, N. Jaitly, and N. M. Shazeer, "Scheduled sampling for sequence prediction with recurrent neural networks," in *Advances in Neural Information Processing Systems, NIPS*, 2015.

[44] A. Saxena, J. Driemeyer, and A. Y. Ng, "Learning 3-d object orientation from images," in *Proceedings of the International Conference on Robotics and Automation*, ICRA, pp. 4266–4272, 2009.