Liquids & Robots: An Investigation of Techniques for Robotic Interaction with Liquids

Connor Schenck

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2018

Program Authorized to Offer Degree:
Computer Science & Engineering

University of Washington

Abstract

Liquids & Robots: An Investigation of Techniques for Robotic Interaction with Liquids

Connor Schenck

Chair of the Supervisory Committee:
Professor Dieter Fox
Computer Science & Engineering

Liquids are an important part of everyday human environments. We use them for common tasks such as pouring coffee, mixing ingredients for a recipe, or washing hands. For a robot to operate effectively on such tasks, it must be able to robustly handle liquids. In this thesis, we investigate ways in which robots can overcome some of the challenges inherent in interacting with liquids. We investigate how robots can *perceive*, *reason* about, and *manipulate* liquids. We split this research into two parts. The first part focuses on investigating how learning-based methods can be used to solve tasks involving liquids. The second part focuses on how model-based methods may be used and how learning- and model-based methods may be combined.

In the first part of this thesis we investigate how deep learning can be adapted to tasks involving liquids. We develop several deep network architectures for the task of *detection*, a liquid *perception* task wherein the robot must label pixels in its color camera as liquid or not-liquid. Our results show that networks able to integrate temporal information have superior performance to those that do not, indicating that this may be necessary for the perception of translucent liquids. Additionally, we apply our network architectures to the related task of *tracking*, a liquid *reasoning* task where the robot must identify the pixel locations of all liquid, seen and unseen, in an image based on its learned knowledge of liquid physics. Our results show that the best performing network was one with an explicit memory, suggesting that liquid reasoning tasks may be easier to solve when passing explicit state information forward in time. Finally, we apply our deep learning architectures to the task of pouring specific amounts of liquid, a *manipulation* task requiring precise control. The results show that by using our deep neural networks, the robot was able to pour specific amounts of liquid using only RGB feedback.

In the second part of this thesis we investigate model-based methods for robotic interaction with liquids. Specifically, we focus on physics-based models that incorporate fluid dynamics algorithms. We show how a robot can use a liquid simulator to track the 3D state of liquid over time. By using a strong model, the robot is able to reason in two entirely different contexts using the exact same algorithm: in one case, about the amount of water in a container during a pour action, in the other, about a blockage in an opaque pipe. We extend our strong, physics-based liquid model by creating SPNets. SPNets is an implementation of fluid dynamics with deep learning tools, allowing it to be seamlessly integrated with deep networks as well as enabling fully differentiable fluid dynamics. Our results show that the gradients produced from this model can be used to discover fluid parameters (e.g., viscosity, cohesion) from data, precisely control liquids to move them to desired poses, and train policies directly from the model. We also show how this can be integrated with deep networks to perceive and track the 3D liquid state.

To summarize, this thesis investigates both learning-based and model-based approaches to robotic interaction with liquids. Our results with deep learning, a learning-based approach, show that deep neural networks are proficient at learning to perceive liquids from raw sensory data and at learning basic physical properties of liquids. Our results with liquid simulation, a model-based approach, show that physics-based models are very good at generalizing to a wide variety of tasks. And finally out results with combining these two show how the generalizability of models may be combined with the adaptability of deep learning to enable the application of several robotics methodologies.

TABLE OF CONTENTS

Page

List of Figures iv

Chapter 1: Introduction 1
- 1.1 Motivation 3
- 1.2 Problem Statement 6
- 1.3 Contributions 9
- 1.4 Overview 10

Chapter 2: Related Work 13
- 2.1 Robotics 13
- 2.2 Computational Fluids 18

Part I: Learning-Based Methods 24

Chapter 3: Learning to Pour: A First Look at Liquid Manipulation 25
- 3.1 Experimental Setup 27
- 3.2 Methodology 29
- 3.3 Evaluation 35
- 3.4 Results 36
- 3.5 Discussion 38

Chapter 4: Learning to See: Teaching Robots to Perceive Liquids 40
- 4.1 Task Overview 41
- 4.2 Simulated Data Set 42
- 4.3 Robot Data Set 47
- 4.4 Learning Methodology 54
- 4.5 Evaluation 61

4.6 Results 65
4.7 Discussion 73

Chapter 5: Learning to See & Pour: Combining Manipulation and Perception . . . 76
5.1 Technical Approach 77
5.2 Experiments & Results 88
5.3 Conclusion and Future Work 96

Chapter 6: Learning to See Through Objects: Reasoning about Occluded Liquid . 98
6.1 Task Overview 99
6.2 Data Set 99
6.3 Learning Methodology 101
6.4 Evaluation 104
6.5 Results 106
6.6 Discussion 109

Part II: Incorporating Fluid Physics Models 112

Chapter 7: From 2D to 3D: Reasoning About Liquids Using Liquid simulators . . 113
7.1 Open-Loop Liquid Simulator 115
7.2 Closed-Loop Liquid Simulators 118
7.3 Experimental Setup 123
7.4 Experiments & Results 129
7.5 Discussion 135

Chapter 8: SPNets: Combining Deep Learning with Liquid Simulation 139
8.1 Position Based Fluids 141
8.2 Smooth Particle Networks 144
8.3 Evaluation & Results 150
8.4 Combining Perception and SPNets 161
8.5 Discussion 169

Chapter 9: Conclusion 172
9.1 Summary 172
9.2 Discussion 174

9.3 Future Work 177

Bibliography 179

LIST OF FIGURES

Figure Number Page

1.1 Outline of the problems studied in this thesis. The columns, *perception*, *reasoning*, and *manipulation*, comprise each of the problems in logical order. Each column lists the tasks we explore in this thesis as it pertains to that problem. The dashed boxes indicate which tasks are covered in each chapter (best viewed in color). 2

3.1 The robot used in these experiments. It is a Rethink Robotics Baxter Research Robot, equipped with two 7-DOF arms. Also shown is the experimental setup with the cup in the robot's gripper positioned above the table, with the bowl resting on top of the scale. 27

3.2 The neural network used to learn the policy. The input layer is fed into a fully-connected hidden layer, and then the output of the hidden layer is element-wise multiplied by the input layer, which is finally fed into the output layer. 34

3.3 The error in grams after each iteration. The error is how far the robot was from the pouring target at the end of the rollout. 37

4.1 The scene used to simulate pouring liquids. The background sphere is cut-away to show its interior. From left to right: The scene shown without any texture or materials; The background image sphere texture added; The video on the plane added in addition to the background texture; and The scene fully textured with all materials. 42

4.2 The objects used to generate the simulated dataset. The first row are the three source containers. The last row are the 3 target containers. The objects are each shown here with 1 of their possible 7 textures. 44

4.3 Examples of frames from the simulated dataset. The top row is the raw RGB images generated by the renderer; the bottom row shows the ground truth liquid location for visible liquid. 46

4.4 The robot used in the experiments in this chapter. It is shown here in front of a table, holding the *bottle* in its right gripper, with the *fruit bowl* placed on the table. 47

4.5 An example of obtaining the ground truth liquid labels from the thermal camera. From left to right: The color image from the RGBD camera; The thermal image from the thermal camera transformed to the color pixel space; The result after thresholding the values in the thermal image; and An overlay of the liquid pixels onto the color image. 48

4.6 The objects used to collect the dataset. The first row are the three source containers. The last two rows are the 6 target containers. 50

4.7 The target containers used to create the testing set. 53

4.8 Different images of the same frame from the same sequence. The upper part of this figure shows the different types of network inputs (RGB, grayscale, optical flow, RGB+optical flow, and grayscale+optical flow). The lower part shows the desired network outputs (pixel labels for *liquid* and *not-liquid*). . . 55

4.9 The 3 different types of input blocks. The first is used when the network takes only a single type of input; the second two are used when combining two different types of input. Here gray boxes are the feature representations at each level of the network, and the colored squares are the layers that operate on each representation. Gray boxes immediately adjacent indicate channel-wise concatenation. 58

4.10 The three types of networks we tested. The first is a standard FCN. The second is an FCN that takes in a series of consecutive frames. The final is a recurrent network that uses an LSTM layer to enable the recurrence. As in Figure 4.9, the gray boxes are the feature representations at each level of the network, and the colored squares are the layers that operate on each representation. Gray boxes immediately adjacent indicate channel-wise concatenation (the dashed line in the MF-FCN indicates a concatenation over the range of inputs). N_I indicates the size of the output of the input block, with $N_I = 2$ for the late-fusion blocks and $N_I = 1$ for all other blocks. The LSTM-FCN takes its own output from the previous timestep as input (lower-left), convolves it through 3 layers, and concatenates it with the output of the input block. The LSTM layer is implemented using the layout described in Figure 1 of [41] 60

4.11 Example frames from the 3 network types on the *detection* task on the simulated dataset. The sequences shown here were randomly selected from the test set and the frame with the largest amount of liquid visible was selected. The last sequence was selected to show how the networks perform when no liquid is present. 66

4.12 Precision-recall curves for the simulated dataset. The first three show the curves for the three network types on the *fixed-view* subset. The last graph shows the performance of the LSTM network on the *multi-view* subset. The different lines show the different amounts of slack, i.e., how far a positive classification can be from a true positive to still count as correct. The area under the curve (AUC) is shown for the 0 slack curve. 67

4.13 The area under the curve (AUC) for the precision-recall curves for the networks for the *detection* task on the robot dataset. The top table shows the AUC for the single-frame FCN; the bottom shows the AUC for the LSTM-FCN. The tables show the AUC for different types of input, with rows for different types of image data (RGB or grayscale) and the columns for different types of optical flow (none, early-fusion, or late-fusion). Each cell shows the AUC on the train set (upper) and the AUC on the test set (lower), all computed with 0 slack. 69

4.14 Example frames for the LSTM-FCN for the *detection* task on the robot dataset with different types of input images. The first row shows the color image for reference and the row immediately below it shows the ground truth. All these images are from the test set with target object not seen in the train set. The last row shows the output of the baseline. 70

4.15 The area under the curve on the test set at each iteration of training. The red line shows the performance of the LSTM-FCN trained solely on the robot dataset; the blue line shows the performance of the LSTM-FCN initialized from the simulated dataset. 72

5.1 The Baxter robot used in our experiments. In its right gripper it holds the cup used as the source container. On the table in front of the robot are the three target containers (from left to right): the small bowl, tan mug, and redgray mug. 77

5.2 The entire robot control system using the recurrent neural network for detections and the multi-frame network for volume estimation. The recurrent detection network (top) takes both the color image and its own detections from the previous time step and produces a liquid detection heatmap. The multi-frame network (center) takes a sequence of detections cropped around the target container and outputs a distribution over volumes in the container. The output of this network is fed into a HMM, which estimates the volume of the container. This is passed into a PID controller, which computes the robot's control signal. 78

5.3 An example frame from a pouring sequence. The left image shows the color camera from the robot's perspective, the left-middle image shows a heatmap of the thermal camera after it has been registered to the color camera, the right-middle image shows the water labeled by thresholding the thermal image, and the right image shows the output of the detection network based on the RGB input. 79

5.4 Diagram of the camera looking into the target container. The blue shows where the liquid is expected to be, given a volume and the corresponding fill height, h_t. The blue lines show pixels expected to be classified as *liquid* and the orange lines show pixels that are expected to be classified as *not-liquid*. The gray shows pixels outside the container and not used by our algorithm. . 82

5.5 The plot shows the scale reading compared to the thermal camera with the model based method for each of the target containers. 89

5.6 Root mean squared error in milliliters of each of the methods for volume estimation. The left plot shows the error for the model-based method using either the thermal image or the output of the detection network as pixel labels. The right plot shows the error for the model-free methods when the networks take as input only the liquid detections (red), the liquid detections plus the color image (green), and the liquid detections plus the depth image (blue). . 90

5.7 The volume estimates for the two model-based methods and the multi-frame detection only model-free network. The black dashed line is the baseline ground truth. We randomly selected one sequence for each target container from our test set to display here. Best viewed in color. 91

5.8 Plot of the result of each pour using our model-free method as input to the controller. The x-axis is the target amount that the robot was attempting to reach, and the y-axis is the actual amount the robot poured as measured by an external scale. The points are color-coded by the target container. The black dashed line shows a 1:1 correspondence for reference. 93

5.9 Reference images for each of the three target containers. This is exactly the perspective the robot sees when looking at the containers. Notice that a 50ml difference is difficult to perceive even for a human (best viewed enlarged). . . 94

6.1 Examples of frames from the simulated dataset. The left column is the raw RGB images generated by the renderer (for reference); the center column shows the ground truth labeling output by the simulator; the right column shows the ground truth liquid location for all liquid in the scene. 100

6.2 Different images of the same frame from the same sequence. The upper part of this figure shows the different types of network inputs (RGB, semantic labels, and grayscale combined with optical flow). The lower part shows the type of desired network output (all liquid). 102

6.3 The precision-recall curves for the *tracking* task on the simulated dataset. The first 3 show the performance of the three network types on the *tracking* task alone. The last graph shows the performance of the LSTM-FCN on the combined *detection* & *tracking* task using the standard RGB format as input. The different lines show the different amounts of slack, i.e., how far a positive classification can be from a true positive to still count as correct. The area under the curve (AUC) is shown for the 0 slack curve. 107

6.4 The performance on the combined *detection & tracking* task of the LSTM-FCN that takes as input RGB images compared to the performance of the LSTM-FCN that takes as input grayscale images plus optical flow using the early-fusion input block. The upper graph shows the precision-recall plot for both networks on both the train and test sets. The lower table shows the corresponding area under the curve for each curve. Note that here for clarity we don't use any slack unlike in figures 6.3 (equivalent to a slack of 0). . . . 108

7.1 A comparison between open-loop and closed-loop liquid modeling. The left-most figure shows the color image of the scene for reference and the left-center figure shows the same image with the actual liquid pixels labeled. The right two images show the color image, but with the liquid from the simulator shown. 114

7.2 Objects used during the experiments. The top row shows the two containers the robot poured from as well as the pipe junction. The leftmost bowl in the bottom row was used in the pouring and the right two were used during the pipe junction experiments. 124

7.3 The 3 types of blockages placed in the pipe junction. (left to right) Pipe junction with no blockage; left leg is partially blocked; and left leg is fully blocked. 125

7.4 Acquiring liquid labels from the thermal camera. The leftmost image is a color image of the scene, the center-left shows the corresponding thermal image transformed to the color image's space. The center-right image shows the liquid labels acquired via thresholding the thermal image, and the rightmost shows the labels overlayed on the color image. 127

7.5 The table shows the IOU for each method. The graph shows the IOU at every timestep across one of the pouring experiments (*bottle* filled to *30%*). 129

7.6 Probability distribution over the estimated initial fill amounts. They are aggregated by the true fill amounts. From left to right and top to bottom they are empty, 30% full, 60% full, and 90% full (indicated by the *). The blue bars show results from the open loop method, cyan for the MAP filter, and red for the warp field. 131

7.7 Probability distribution over the blockage location over time for a single interaction. The 5 diagrams across the top correspond to the five different simulations the robot ran, each color-coded to the corresponding line in the plot. The true blockage was placed in the left leg and only partially blocked the leg (in the keys in the top row, second from right). Best viewed in color. 134

8.1 The PBF algorithm. P is the list of particle locations, V is the list of particle velocities, and ΔT is the timestep duration. 140

8.2 The layout for SPNet. The upper-left shows the overall layout of the network. The functions SolvePressure, SolveCohesion, SolveSurfaceTension, and SolveObjectCollisions are collapsed for readability. The lower-right shows the expansion of the SolveObjectCollisions function, with the line in the top of the box being the input to the SolveObjectCollisions in the upper-left diagram and the line out of the bottom of the box being the line out of the box. The lower-left shows the expansion of the SolvePressure function. For clarity, the input line (which represents the particle positions) is colored green and the line representing the particle pressures is colored pink. 148

8.3 (Top) The *ladle* scene. The left two images are before and after snapshots, the right image shows the particles projected onto a virtual camera image (with the objects shown in dark gray for reference). (Bottom) The difference between the estimated and ground truth fluid parameter values for cohesion λ_c and viscosity λ_v after each iteration of training for both the L1 loss and the projection loss. The color of the lines indicates the ground truth λ_c and λ_v values. The cohesion parameter λ_c was initialized to 0.1 in all cases except when the ground truth was 0.1, in which case it was initialized to 0.05. The viscosity parameter λ_v was initialized to 60 in all cases except when the ground truth was 60, in which case it was initialized to 30. 152

8.4 The control scenes used in the evaluations in this chapter. The top row shows the initial scene; the bottom row shows the scene after several seconds. . . . 154

8.5 Diagram of the rollout procedure for optimizing the controls u. The dynamics are computed forward (black arrows) for a fixed number of timesteps into the future (shown here are 3). Then gradients of the loss are computed with respect to the controls backwards (blue arrows) through the rollout using backpropagation. 156

8.6 Results from the liquid control task. From left to right and top to bottom: The *plate* scene. The numbers in each bowl indicate the percent of particles that were successfully placed in that bowl when that bowl was the target. The *pouring* scene. The x axis is the targeted pour amount and the y axis is the amount of liquid that was actually poured where the red marks indicate each of the 11 pours. The *catching* scene. Shown is the percent of liquid caught by the target cup where the rows indicate the initial direction of movement of the source. 157

8.7 Diagram of the rollout procedure for optimizing the policy parameters θ. This is very similar to the procedure shown in Figure 8.5. The dynamics are computed forward (black arrows) for a fixed number of timesteps into the future (shown here are 3). Then gradients of the loss are computed with respect to θ backwards (blue arrows) through the rollout using backpropagation. 159

8.8 A diagram of the SolvePerception method. It takes as input the current particle state (center-left) and the current RGB image (lower-left). The RGB image is passed through the LSTM-FCN to generate pixel labels, the particle state is projected onto 2D. From this, 2 gradient fields are computed, where the gradient points in the direction of the closest *liquid* pixel. These fields are then blended and projected onto the particles in 3D. 164

8.9 Diagram of the rollout procedure for optimizing the parameters of the perception network θ. This is very similar to the procedures shown in Figures 8.5 and 8.7. The dynamics are computed forward (black arrows) for a fixed number of timesteps into the future (shown here are 3). In this case the controls u are fixed. The gradients of the loss are computed with respect to θ backwards (blue arrows) through the rollout using backpropagation. The box "Perception Correction" corresponds to the network shown in Figure 8.8 and θ are its parameters. Note that here the inner loop of the PBF algorithm in SPNets is shown (with 3 iterations) since this is how the perception correction is intergrated into the dynamics. 166

8.10 Results when combining SPNets with perception. The images in the top and bottom rows show 2 example frames. From left-to-right: the RGB image (for reference), the RGB image with SPNets overlayed, and the RGB image with SPNets with perception overlayed. In the overlays, the blue color indicates ground truth liquid pixels, green indicates the liquid particles, and yellow indicates where they overlap. 167

ACKNOWLEDGMENTS

I would like to thank my advisor Professor Dieter Fox, who provided invaluable guidance throughout my research. Dieter helped me to discover the topic of liquids and robots, and helped me determine not only what would and wouldn't work, but also what would constitute a valuable contribution to the field of robotics research. I would also like to thank my committee members Professor Maya Cakmak, who helped advise me during my initial years of research, Professor Siddhartha Srinivasa, and graduate school representative Professor Samuel Burden.

I would also like to thank my peers in graduate school and at some of the companies I interned at while conducting this research. My fellow graduate students and postdocs in the Robotics & State Estimation Lab (listed in no particular order) Daniel Gordon, Aaron Walsman, Arunkumar Byravan, Tanner Schmidt, Yuyin Sun, Luis Puig, and Yu Xiang for helping me with many of the technical aspects of this research and also providing guidance on many research ideas. I want to thank Sudharsan Prabu for helping to collect much of the data used in this research. I also appreciate the input of Jonathan Tompson of Google for helping me to understand deep learning better and Miles Macklin of Nvidia for helping me with Nvidia FleX.

Finally, I want to acknowledge the support I've received from several funding institutions. Much of the research in this thesis was funded by the National Science Foundation either via the Graduate Fellowship Research Program or a National Robotics Initiative grant, and I greatly appreciate their support. I also appreciate the support of Nvidia, who funded me via an internship for part of this research and also allowed me access to both their Nvidia FleX software and team of experts who created it. I also want to thank the University of

Washington Paul G. Allen School of Computer Science & Engineering for their support and for providing me access to amazing researchers and facilities.

I would like to make one final acknowledgement. I want to acknowledge a chocolate cake. Yes, I really do want to acknowledge the contribution of a chocolate cake to my research, for it made a significant one. Growing up, my parents would bake a chocolate cake for my birthdays or other special events. However, when I moved far from home to begin graduate school, this was no longer possible. A normal person in this situation would just learn to bake one on their own, but I am not a normal person; I am a roboticist and I automate things for a living. So instead I endeavoured to design a robot that could make such cakes for me. It was during this process that Dieter and I discovered the relative dearth of research around liquids and robots. We decided to focus on this research area because it was something important for robots to understand if they ever wanted to solve the cake baking task, or for that matter a wide variety of other household tasks. So I really do want to thank a chocolate cake, because the desire for it set in motion all the research in this thesis.

DEDICATION

This thesis is dedicated to Dr. Schutte.

Chapter 1
INTRODUCTION

Liquids are everywhere. They cover 70% of our planet's surface. Life began in liquid [33], and liquids are necessary for life to exist at all [7]. Humans use liquids everyday in a variety of tasks from taking a shower to drinking coffee to washing dishes. There are even large plumbing infrastructures in place to transport liquids to and from homes, offices, and other buildings. It is clear that liquids play a vital role in everyday life. Thus any robotic system that is designed to assist humans in common household tasks must be able to effectively deal with liquids. For example, a cooking robot must be able to handle liquid ingredients, a robot washing dishes must be able to handle liquid cleaner, and a robot server must be able to transport open-top full containers without spilling.

In this thesis, we investigate a framework to enable robots to intelligently handle liquids. We break this framework down into three steps: 1) First the robot must *perceive* its environment to locate the liquid. 2) Next the robot must *reason* about the liquid it sees in order to infer relevant information (e.g., if a container may be full of liquid). 3) And finally the robot must use this information to *manipulate* the liquid to a desired configuration. Figure 1.1 shows this framework. We investigate several tasks for each of the steps of our framework, which are shown in the figure. We specifically selected a more traditional model-based approach with individual steps of the pipeline specified beforehand, as opposed to a fully end-to-end opaque framework, because it allows us to analyze each step individually. By doing so, we can inspect the components of each step in order to gain valuable insights.

The research done in this thesis is primarily an *investigation* into how robots might *perceive*, *reason* about, and *manipulate* liquids. For the tasks we examine here, we look at both how we can adapt existing methodology for other tasks to liquids as well as how we can

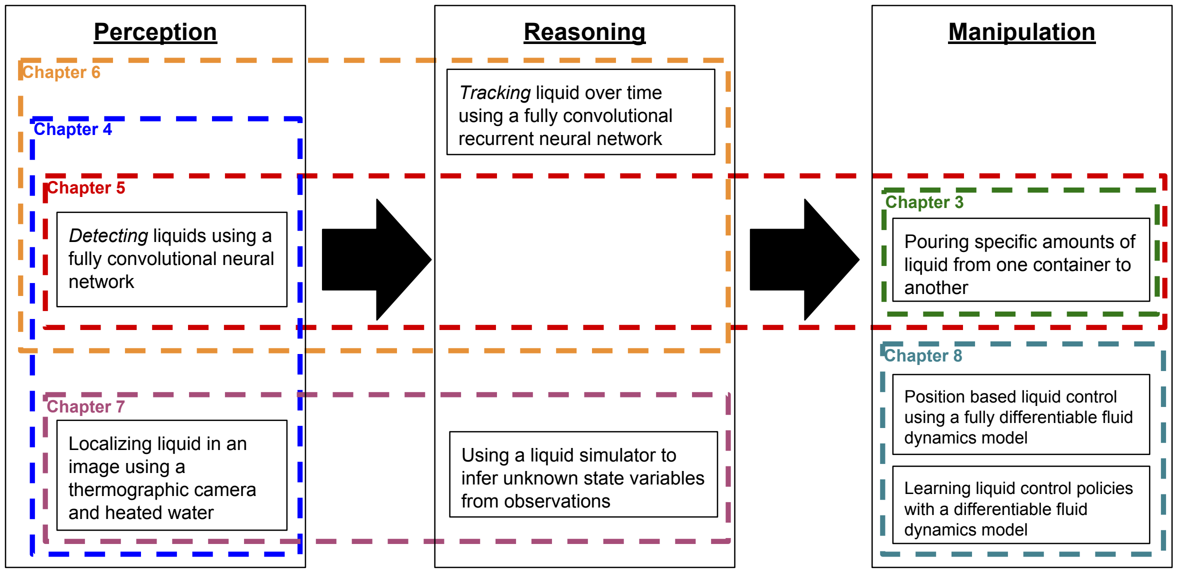

Perception
Reasoning
Manipulation
Chapter 6
Chapter 4
Chapter 5
Detecting liquids using a fully convolutional neural network
Tracking liquid over time using a fully convolutional recurrent neural network
Chapter 3
Pouring specific amounts of liquid from one container to another
Chapter 8
Position based liquid control using a fully differentiable fluid dynamics model
Learning liquid control policies with a differentiable fluid dynamics model
Chapter 7
Localizing liquid in an image using a thermographic camera and heated water
Using a liquid simulator to infer unknown state variables from observations

Figure 1.1: Outline of the problems studied in this thesis. The columns, *perception*, *reasoning*, and *manipulation*, comprise each of the problems in logical order. Each column lists the tasks we explore in this thesis as it pertains to that problem. The dashed boxes indicate which tasks are covered in each chapter (best viewed in color).

extend and develop our own methods to add to the robot's repertoire. We develop several systems in this thesis including a deep learning architecture for perceiving liquids, a method for precise pouring of liquids from raw RGB feedback, and a method for integrating deep neural networks with fluid dynamics. Our results reveal multiple insights about how robots can interact with liquids, such as the importance of integrating temporal information on the difficult perception task or the reasoning power of 3D physics-based models for inferring unknown information about liquids. We believe that the conclusions and insights in the following pages of this thesis constitute a significant contribution to our understanding of how robots can interact intelligently with liquids.

1.1 Motivation

Interacting with liquids is not a trivial task, but humans have been doing it for a long time. In fact humans interact with liquids from a young age. Studies have shown that even infants can distinguish between rigid objects and “substances”, or liquids [51]. They have also shown that infants as young as five months have knowledge about how substances behave and interact with solid objects [50]. Furthermore, infants as young as 10 months have the ability to distinguish quantities of non-cohesive substances as greater or less than, although the quantity ratio must be larger for substances than for solid objects, suggesting that humans use a different mechanism to quantify substances than to quantify objects [154]. This supports the claim that liquids are important to human environments since even young infants have at least some capacity to reason about them. Thus if liquids are so important to humans, then they must also be important for robots wishing to operate alongside humans. Furthermore, other studies have shown a correlation between humans’ understanding of fluid dynamics and what would be expected of a probabilistic fluid physics model [8], which suggests that humans have more than a simple perceptual understanding of fluid physics. This implies that liquids have their own reasoning pathways in the brain, suggesting that generic rigid object reasoning is not sufficient for handling liquids. Thus for robots to operate in human environments, they may need reasoning capabilities beyond just rigid body physics. In this thesis, we investigate ways to enable robots to have these capabilities.

Robots in industrial settings have been performing tasks involving liquids for many years [55, 109, 85, 86, 78, 157]. So if robots can already handle liquids, why investigate these capabilities any further? The primary drawback to industrial approaches is that these robots are often given highly constrained tasks and specialized hardware, obviating the need for any precise understanding of the liquids they work with. For example, a pancake batter dispensing robot suspended from a gantry above a griddle may only know that it needs to open and close its valve on a fixed schedule without any knowledge of how the batter is released onto the griddle. This is acceptable in a highly constrained task environment such as a factory. However, this is not feasible for a robot in an unstructured environment such as a kitchen. For the robot to make a pancake, it would need to know the relative locations of the objects and be able to monitor and control the flow of batter onto the griddle to create the desired shape. In the research here we are interested in robots that can interact in everyday human environments. Single-purpose robots are only useful in factories, where economies of scale apply; but in unstructured environment like homes they are uneconomical. Instead, we must rely on more general-purpose robots, which cannot rely on the structure imposed in a factory and instead must be able to understand their environment on their own. For this reason, we focus here on robotic interaction with liquids in unstructured environments and the capabilities such a robot must have.

To better understand exactly what capabilities a robot in this kind of environment would need to handle liquids, let us consider a prototypical task: baking a cake. The first step involves mixing together eggs, wet ingredients such as heavy cream and vanilla, and dry ingredients such as flour and sugar. To do this, the robot must be able to 1) localize the objects such as the mixing bowl, eggs, and containers with the ingredients, 2) accurately perceive the location and amounts of ingredients, 3) reason about the results of manipulating the objects, and 4) perform precise, closed-loop actions to manipulate those objects and their ingredients. 1) has been well-studied in the robotics and computer vision literature [84, 160, 20, 53, 59, 119, 147, 115, 158]. 3) and 4), at least for rigid objects, have also been studied by prior work in [74, 73, 63, 95, 36, 168, 9, 61] and [30, 110, 138, 82, 5, 24, 28] respectively. However there has been relatively little research relevant to 2) or to 3) and 4) as it pertains to liquids such as heavy cream or granular media such as flour. We go into detail on what research there is pertaining to those areas in chapter 2

The next step, after thoroughly mixing all the ingredients, is to pour the cake batter into a pan. While robotic pouring has been examined in various contexts [79, 18, 108, 144, 127], all of this work has focused on the general task of moving liquid from one container to another. Pouring cake batter into a pan differs from the general pouring task in that it is necessary that the cake batter in the pan be evenly distributed, which can be done either via moving the source container to distribute the batter or by applying a spoon after. Either way, it requires the robot to have a detailed understanding of the state of the liquid and how to precisely manipulate it. After this the cake pan needs to be placed in the oven for a fixed amount of time. Grasping and moving rigid objects subject to constraints (in this case the pan must remain upright) has much prior work [61, 48, 4, 139, 10, 70]. Once the cake has been removed from the oven after baking, the last step is to frost it. Creating the frosting is very similar to creating the cake batter (mixing dry and wet ingredients together) and requires largely the same skills. However, spreading the frosting requires that the robot be able to perceive and manipulate via a tool a highly viscous fluid (the frosting) in order to spread it over an uneven surface (the cake). There has been one work that has looked at using a tool to manipulate granular media [161], although they focused on the related but different task of cleaning.

What is missing? What capabilities do modern robots lack in order to bake a cake in an unstructured kitchen? While there has been a lot of research directly applicable to many components of the task such as localizing objects or grasping and moving objects, there has been a relative dearth of research as it pertains to the skills necessary for robots to manipulate liquids and granular media. This is not unexpected; liquids and granular media are *hard*. They're non-rigid, which means a tool or container is required to manipulate them; a robot can't just grasp them directly. They're non-rigidity also means that their state space is high dimensional, unlike rigid objects who's state can be represented by a 6 degree-of-freedom translation and rotation. Furthermore, they are simply more dangerous to robots than rigid objects; a grain of sand in a joint or a splash of water on a circuit can easily damage an expensive robot. That is not to say there has been no research that has attempted to work around these limitations for cooking tasks. Both [169] and [98] examined ways for robots to hold and break eggs, and [40, 14, 74] attempted to create a full robotic cooking system, however [40, 14] only reasoned about the tasks at an abstract level, leaving hand-tuned primitives to carry out the low-level functions, and [74] only performed cooking in simulation with a course approximation for non-rigid substances, making it difficult to transfer to a real robot. Indeed, there has been very little work to this point that has investigated how a robot can fill in the gaps in its cake-baking skillset to allow it to manipulate liquids and granular media in an intelligent manner.

While both granular media and liquids are interesting areas for research and are necessary to fully enable a robot to bake a cake, in order to focus this thesis we look at only liquids here. Given the similarities between liquids and granular media (e.g., non-rigidity, high-dimensional state space), some of our research may also be applicable to granular media, although here we only focus on liquids due to many of the unique problems posed by them (e.g., many liquids are translucent, which many granular media are not). However, in our concurrent work [134] we investigate one method for enabling robots to manipulate granular media. One goal for future work is to determine how our methods in this thesis may be applied or extended to granular media.

1.2 Problem Statement

In this thesis we focus on studying the capabilities that a robot would need to solve problems similar to the cake baking task described in the previous section. That is, tasks where the robot is placed in relatively unstructured environment and must use liquids with human-level precision to solve tasks. Specifically, we consider the baseline setup to be a robot with an arm and attached gripper placed in front of a tabletop[1]. The robot is then given a task and must use its suite of sensors to perceive its environment and decide on what actions to take. We primarily focus on the task of *pouring*, since it is such a common task involving liquids, however we also consider other tasks as well. We break down our tasks to focus on the different components of liquid manipulation that a robot must solve in order to complete the tasks. We look at how robots can *perceive* liquids, *reason about* liquids, and finally

[1]While there are many interesting tasks that involve mobile robots and liquids (e.g., carrying an open-top container without spilling), in this thesis for simplicity we consider only a stationary robot manipulating liquids over a tabletop.

manipulate liquids. Figure 1.1 shows each of these components and the corresponding tasks we investigate.

Perceiving Liquids: The task of perceiving liquids is a challenging one. Common liquids such as water are translucent, making them difficult to perceive with a color camera and meaning they often scatter the light from infrared-based depth cameras in a difficult-to-predict manner. Furthermore, due to their non-rigidity, liquids need to be contained in other often opaque objects, obscuring much of the liquid from view. However, in recent years deep learning has shown a lot of success solving difficult perception tasks [72, 29, 38, 47, 82, 87]. This strongly suggests that it may be possible to perceive liquids as well.

In this thesis, we examine ways in which we can apply these advances in deep learning to perceiving liquids. We evaluate deep neural networks on the task of *detection*. *Detection* is essentially the task of finding where the liquid is in the robot's perceptual data stream. Here we focus primarily on vision as the main sensor, so *detection* in the vision domain is determining whether each pixel in an image contains liquid or not. That is, the robot's task is to label each pixel in its camera image as *liquid* or *not-liquid*. This task is well-suited for analyzing robotic perception of liquid as it requires solving all the difficult parts of liquid perception while also focusing only on the perception without conflating it with other tasks. In chapter 4 we analyze how deep learning methods can be applied to solve this task.

Reasoning About Liquids: Reasoning about liquids is also quite difficult. Their non-rigidity means that to accurately represent the full state of the liquid it must be relatively high-dimensional, on the order of thousands to tens of thousands of degrees of freedom (DOF). But many robotic algorithms are only designed with lower-dimensional state representations in mind and have difficulty extending to such high DOFs [146, 83, 24, 80]. One solution studied in the literature is to summarize the liquid state in a few variables (e.g., amount of liquid in a container) so that it can be used with traditional robotics algorithms [164, 165, 79]. However, the drawback to this method is that it limits the reasoning capability of the robot to being only task specific or to only applying at a very high-level. This is problematic if a robot wishes to perform fine-grained reasoning about liquids.

In this thesis, we examine how robots can use high-dimensional liquid state representations to reason about liquids. We define *reasoning about* liquids to mean inferring any unknown about the liquid that is not directly observable or given by the problem. For example, the robot may infer from seeing the surface of a liquid in a container that the container below that point is also filled with liquid since it may know (or learned) that liquid is heavier than air and that no air can exist under it. Or, given the liquid's current state, a robot may use its knowledge of liquid physics to infer future states. The goal of reasoning about liquids is to use knowledge a robot gained elsewhere about liquids and their behavior (either through learning or a model) to "fill in" unknowns in the robot's environment. In chapter 6 we examine the task of inferring the liquid state in 2D (using pixels as the state representation) given only the observation. In chapter 7 we examine the task of using a full liquid dynamics model to determine unobserved state variables in the environment by comparing the model to observations.

Manipulating Liquids: The final component we examine in this thesis is *manipulating* liquids. This is also difficult, in part due to its reliance on the previous two but also due to difficulties related directly to manipulation of liquids. Due to its necessarily high-dimensional state, liquid manipulation has a tendency to be chaotic, with small deviations amplified by the many degrees of freedom. For example, a slight alteration in a pouring trajectory can easily result in spilled liquid, or a slight over-torque on a mixing instrument can cause liquid to splash to undesirable locations. Additionally, liquid interactions are often non-reversible, e.g., when pouring the liquid can travel from the source to the target but not the other way. Finally, manipulating liquids can be dangerous for robots. Both [44] and [14] covered their robots in plastic to prevent damage from liquids.

In this thesis, we examine tasks where the robot must perform precise control tasks with liquids. A precise task is one in which the robot must exhibit a deeper understanding of the liquid and its physical properties. For example, in a pouring task, if the goal is simply to move all contents from one container to the other, then it can be specified with constraints on the orientation of the containers and a reward for holding one upended over the other, which requires no understanding of the underlying contents of the container. Alternatively, if the goal is to move a specific amount of liquid from one container to the other, the robot must have some understanding of how the liquid is flowing from one container to the other in order to transfer the correct amount. We look at tasks like this; tasks that require the robot to have at least some understanding of the liquids involved so that it can precisely control them. In chapters 3 and 5 we examine the task of pouring precise amounts of liquid and in chapter 8 we examine moving liquid with rigid objects to desired goal configurations.

1.3 Contributions

Our focus in this thesis is on taking methods developed in robotics, computer vision, machine learning, and fluid mechanics and applying them to robotic liquid manipulation tasks. This is not as straightforward as it might seem. As stated in previous sections, liquids have many qualitative differences with rigid objects, for which many of these methods were developed. As such, in this thesis, much of the work focuses on adapting and modifying many existing methods to be suitable for tasks involving liquids. We evaluate various ways to apply existing methodology to liquid tasks, ways to modify existing methodology for liquid tasks, and even develop our own methodology to combine different approaches. Our main contributions are as follows:

- **Evaluation of Deep Learning on Liquid Perception:** We thoroughly evaluate many variations of deep convolutional neural networks on liquid perception tasks. We compare multiple architectures, training regimens, and input formats. We test them on the *detection* task using water and report results on both a randomly selected test set and a test set containing unseen objects. Our results show that the best performance is achieved by networks that can integrate temporal information over time. This strongly suggests that in order to perceive translucent liquids, it is necessary to aggregate changes in visual appearance such as light refractions or reflections over time.

- **Application of Closed-Loop Control for a Liquid Control Task:** We also show how these methods can be used to perform a closed-loop liquid control task. We demonstrate the first instance in the literature of a robot utilizing a deep neural network in order to precisely control liquids. Our experiments show how models learned from data can be used to successfully pour specific amounts of liquid using only a color camera. This suggests that learning-based approaches to liquid control can learn to perform tasks with human-level accuracy.

- **Using a Closed-Loop Fluid Simulator for Liquid Reasoning:** We adapt standard fluid dynamics models for robotics and show how they can be used by a robot to reason about liquids. We develop a way to use relatively simple perceptual models combined with a fluid simulator to track the state of the liquid over time. We then demonstrate how this can be used by a robot to perform reasoning tasks about its environment such as determining how full a container is or where a hidden obstacle might be. We show the versatility and generalizability of model-based methods for full 3D reasoning about liquids.

- **Combining Fluid Simulation and Deep Learning:** Finally, we show how we can combine our learning-based methods that utilize deep networks with our model-based methods that utilize fluid simulators. We propose SPNets, a framework that integrates deep learning with fluid dynamics. We show how this can enable a robot to reason about and control liquids robustly.

1.4 Overview

This thesis is split into two parts: the first part focuses on learning-based methods, and the second focuses on model-based methods. The split in this thesis reflects the split in the field of robotics in general. Traditional robotics methods rely on models of their environment to operate, and they have shown considerable success in doing so [146, 64, 32, 150, 81, 136]. However, recently much work has shown how model-free learning-based methods, primarily utilizing deep learning, can solve many difficult robotic tasks [82, 47, 34, 110, 124, 57]. This has led to a split in the robotics literature between model-based methods and learning-based (or model-free) methods. Both have their advantages and disadvantages. Learning-based methods do not require pre-specified models and can adapt very well to a wide-variety of tasks, however they tend to require large amounts of training data and have difficulty generalizing outside of the trained task. Model-based methods, on the other hand, tend to generalize better and don't usually require copious amounts of training data, although they require models to be specified in advance and have difficulty adapting to differences between the model and the robot's environment. In this thesis, we examine both types of methods and then in chapter 8 we propose a way to combine both for liquid tasks. Figure 1.1 shows which tasks we investigate in each chapter.

The rest of this thesis is laid out as follows. The next chapter, chapter 2, goes into detail about prior work done in robotics, computer vision, machine learning, and fluid mechanics that is related to robotic manipulation of liquids. Following this chapters 3 through 6 detail our evaluation of learning-based methods to robotic tasks involving liquids. In chapter 3 we start our examination with a simple liquid manipulation task: pouring with direct instrumentation of the environment. We place a real-time scale under the target container to directly measure liquid flow and we apply guided policy search [83] to train the robot to pour precise amounts of liquid. However, in a real environment a robot cannot always rely on direct instrumentation such as a scale, so the next step is to examine ways to use a more general purpose sensor. In chapter 4 we do just that, looking at how a robot can use a color camera to perceive liquids. We apply deep learning to the task of *detection*, i.e., labeling pixels as *liquid* or *not-liquid*. We evaluate multiple deep architectures on both simulated and real datasets. In chapter 5 we return to the task of pouring, this time utilizing the deep networks developed in the previous chapter for perception instead of a scale. In chapter 6 we further evaluate our deep networks on a liquid reasoning task. We apply the networks to the task of *tracking*, that is, tracking all the unseen liquid in an image given only the visible liquid.

Chapters 7 and 8 detail our evaluation of model-based method for robotic tasks involving liquids. In chapter 7 we examine ways in which a fluid simulator can be useful for a robot operating in a real environment. We simulate the fluid state alongside the real state, using simple perceptual models to “correct” deviations from the real liquid. This is then used by the robot to perform reasoning tasks about its environment, such as how full an unobserved container is or where a hidden object might be. In chapter 8 we propose SPNets, an extension of a fluid simulator that allows it to directly interface with deep neural networks. We examine how it can be used to reason about various liquid properties, solve liquid control tasks, and integrate with the liquid perception deep networks from chapter 4. Finally, we finish this thesis in chapter 9 with a discussion of the results of our experiments and avenues for future work.

Chapter 2
RELATED WORK

The work in this thesis combines ideas from both robotics and computational fluids/fluid dynamics. We draw on techniques in computer vision and robotics to both perceive and manipulate liquids. We also use methodology from computational graphics and fluid dynamics to represent liquids and to implement their dynamics. By combining these, we can develop tools for robots to perceive, reason about, and manipulate liquids. The rest of this chapter is outlined as follows. The following section details related work in robotics pertaining to both perceiving liquids and manipulating them. The section after that describes related work in computational fluids, about both fluid dynamics models and learning as it relates to those models.

2.1 Robotics

2.1.1 Liquid Perception

Liquids are difficult to perceive, often being clear or at least partially transparent and almost always occluded by the container holding them. As a result, there has not been as much emphasis in the literature on perceiving liquids. Nonetheless, there has been some work in this area. Rankin *et al.* [120, 121] investigated ways to detect pools of water from an unmanned ground vehicle navigating rough terrain. They relied on traditional computer vision approaches (this work was performed prior to the wide-spread popularity of deep learning), detecting water puddles via color features or based on sky reflections. By using stereo color cameras mounted on an unmaned ground vehicle and taking advantage of the unique visual properties of water (e.g., gradient of the saturation-to-brightness ratio as compared to surrounding terrain), they were able to reliably detect pools of water along the robot's path and navigate around them. Work by Yamaguchi and Atkeson [165] also used color features to detect liquids. In this work, they positioned a color camera directly across from a clear plastic cup with a solid color backdrop as a robot poured colored water into the cup. This allowed them to use a color filter to determined the amount of liquid in the cup. However, the focus of this work was on robotic control for pouring liquids, so the perception system relied on relatively simple computer vision techniques.

Yamaguchi and Atkeson followed their work in [165] with work focused more directly on the liquid perception problem. In [163] they utilized stereo optical flow for liquid detection. They placed a pair of calibrated color cameras opposite a cup, and then poured liquid into the cup. During each pouring sequence, they calculated the optical flow in each camera image and then used feature matching to match the flow features between the two cameras, resulting in 3D locations for each feature. By using a known camera pose and some assumptions about the nature of the task (e.g., the liquid stream tends to be above the target cup), they were able to isolate the features associated with the liquid. These features were used to detect the location of the liquid in the environment. Because the model relied primarily on non-learned visual features, it was able to reliably detect a wide variety of liquids including water, cola, and even beans. This was a key component to their model: without a lot of parameters to train, it couldn't overfit, and as a result was able to generalize quite well. On the other hand, however, their results did indicate that the model suffered from a lack of precision, something which a model trained from data could learn to overcome. In chapter 4 we evaluate many variants of a learning-based method on the liquid detection task, including a method that integrates insights from [163].

Similar to Yamaguchi and Atkeson, Do *et al.* [26] also developed a model-based technique for liquid perception, albeit with a different approach. To detect the level of liquid in a container, they utilized the infrared (IR) light from a projected light depth camera. When the camera projects its IR pattern onto the scene, the light is refracted through liquid differently than through air. Do *et al.* took advantage of this by developing an algorithm that computes the expected image given either liquid or air refraction and then compares that to the actual image. By using this technique in combination with a probabilistic state estimate, they were able to successfully determine the level of the liquid in a target container over a short time window. In that paper, the authors took advantage of the *distortions* in the image caused by light refracting through the liquid to detect the liquid, however work by Elbrechter *et al.* took the opposite approach. In [27] they used a depth camera to perceive liquids and detect how they moved in order to discriminate between high viscosity and low viscosity liquids. Unlike Do *et al.*, Elbrechter *et al.* used opaque liquids which reflected the IR light, allowing the depth camera to construct an accurate model of their surface. This was crucial as the change in surface shape of the liquid over time was critical for determining the viscosity of the liquid. By using this method, they were able to reliably distinguish between different types of viscous liquids. Similarly, work by Paulun *et al.* [114] also used liquid shape to discriminate viscosity. They generated liquid splatter sequences by utilizing a liquid simulator, which also allowed them to extract binary *liquid/not-liquid* pixel labels from a simulated camera. They then extracted hand-designed shape features from the binary images and used them to train a model to discriminate liquid viscosity. Their model was able to determine the viscosity with the same reliability as human participants.

Finally, no discussion on robotic liquid perception is complete without at least some mention of the objects that contain them. Liquids can almost never be interacted with directly, and so they spend most of their time in containers. Work by Griffith *et al.* [42, 43] has focused on using behavior-grounded approaches to learning about containers. Specifically, in [44] the robot placed containers under a constant water stream flowing from a faucet into a sink. The robot utilized both auditory and proprioceptive data to learn about each container's properties and discriminate containers from non-containers. Other work on containers for liquids by Mottaghi *et al.* [101][1] focused on determining the volume of containers from color images. In that work, they trained a deep neural network to predict the volume of containers from individual color images as well as the potential amount of liquid contained in the container

[1]Several of the authors on this thesis are also co-authors on [101].

given the object's pose. To do this, they generated a dataset with 3D models associated to each visible container and then utilized a liquid simulator to estimate the resting maximum fill amount of the containers for each pose.

2.1.2 Liquid Manipulation

Liquid manipulation is a complicated and oft-treacherous task. Liquids are highly deformable, chaotic, and a spill can easily damage or destroy an expensive robot. Nevertheless, several groups have undertaken research on liquid manipulation, primarily focusing on the task of pouring. Work by Langsfeld *et al.* [79] used human demonstrations to teach a robot how to succeed at a task involving pouring into a moving container. Cakmak and Thomaz [18] evaluated a robot active learning system on prompting humans to provide useful feedback when learning to perform various tasks including pouring. Okada *et al.* [108] created a system that allowed a robot to perform motion planning for pouring and then verify that the robot performed the correct motions after the fact. Finally, work by Tamosiunaite *et al.* [144] used dynamic movement primitives to capture the goal and shape of a pouring trajectory for a robot learner. These works applied a variety of techniques, from motion planning to human demonstrations, to teach a robot how to pour from one container into another.

However, the research in [79, 18, 108, 144] all focused on the broad motions of pouring (e.g., moving the source over the target, rotating the source, etc.) rather than low-level control or closed-loop feedback. Rozo *et al.* [127] developed a method that used real-time closed-loop feedback to pour precise amounts. Specifically, they utilized human demonstrations to teach their robot how to pour and proprioceptive sensors in the robot's arm to provide feedback on the amount that had been poured out. They were able to successfully teach their robot to use feedback to pour 100 ml from a bottle into a cup. Other work by Kennedy *et al.* [60] also used closed-loop feedback to perform precise pouring. In their experiments, a robot was task with dispensing a precise amount of colored liquid from a source container into a transparent target container. By placing a camera directly across from the target, they were able to use a color filter to determine the height of the liquid, which was then fed into a pouring model that determined the robot's actions. This system allowed the robot to robustly pour precise amounts of liquid. In chapters 3 and 5 we also examine the problem of pouring precise amounts of liquid using a scale as feedback (chapter 3) and visual input as feedback (chapter 5).

Precise dispension of liquids isn't the only concern when pouring however. Another major constraint is the final destination of the liquid, i.e., not spilling liquid outside the target. Yano *et al.* [166] utilized precise liquid models to prevent sloshing during a pouring task. In a follow up work, Sugimoto *et al.* [143] augmented those models to control the surface of the liquid contained in a sprue cap to prevent it from spilling. Later research by Kuriyama *et al.* [75] looked at a similar problem. They utilized a fluid simulator to plan motion trajectories that minimized spillage of liquid in a spoon held in the robot's end-effector. More recent work by Pan and Manocha [111] used a physics simulator with simplified fluid dynamics to plan trajectories that allowed the robot to pour liquid without spilling. In a follow up to their work, Pan and Manocha [112] gathered pouring examples on real data, and then used a neural network to infer the liquid-related parameters, which allowed their trajectory planner to generate more accurate plans for the robot.

Pan and Manocha aren't the only authors to utilize a simulator to prevent spillage. Guevara *et al.* [46] also used approximate fluid dynamics to reason about spillage. In their work, the robot first went through a calibration stage to determine the parameters of the liquid, then an optimization stage to optimize the pouring motion for minimum spillage. Yamaguchi and Atkeson also utilized approximations to reason about liquid spills. In [164] they used small, rigid spheres in place of liquid in a rigid-body physics simulator. The robot generated a motion trajectory using differential dynamic programming (DDP) and then poured the spheres from a source into a target, using temporally decomposed dynamics along with hand-designed features to simplify the problem. They followed this with [162] where the robot solved the pouring task by composing different skills (e.g., grasping, tilting, etc.) in a graph structure, optimizing each using DDP. While both [164] and [162] were entirely in simulation, in [165] they applied their method on a real robotic system, successfully pouring liquids with minimal spillage.

Even though it is primarily real robots that we are concerned with, simulation is nonetheless a powerful tool for reasoning about liquids. Work by Kunze and Beetz [73, 74] took advantage of a robot simulator to reason about potential actions available to the robot. They focused on the task of making pancakes, which involves interactions with rigid objects, deformable objects, and liquids. Similar to Yamaguchi and Atkeson [164], they approximated the liquids as rigid spheres. In the simulator, the robot could repeatedly perform the same task (e.g., pouring pancake batter, breaking eggs) with different action parameters to reason about the effect of the parameters on the outcome of the task. In chapter 7 we utilize a fluid simulator to perform similar reasoning about liquids, although with a focus on hidden state variables rather than action parameters.

2.2 Computational Fluids

2.2.1 Fluid Dynamics

In this thesis, we utilize fluid simulators to model liquids, enabling our robot to both reason about and control them. Here we briefly discuss some of the background and related work in the literature about fluid mechanics and simulation. Liquid simulation and fluid mechanics are well researched in the literature [2, 105]. They are commonly used to model fluid flow in areas such as mechanical and aerospace engineering [52], and to model liquid surfaces in computer graphics [16, 23, 103]. The primary physical equations governing fluid mechanics are the Navier-Stokes equations, developed by Claude-Louis Navier and Goerge Gabriel Stokes in 1822 [148]. Since then, several algorithms have been created to implement the Navier-Stokes equations for fluid simulations. The predominant method is the Eulerian method, a finite element method, where the space is divided into a fixed-size grid, and the fluid is modeled as flow between adjacent grid cells [22]. This method has several advantages. First, it has been extensively studied and has well-formulated and stable mathematical models. Second, the representation of the fluid in a grid format makes is relatively straightforward to interface the fluid representation with a deep neural network using a standard 3D convolutional layer. Finally, there are many commercially available implementations of this algorithm (such as the one included in [11]).

However, there are several drawbacks to the Eulerian method. The biggest issue is the tradeoff between resolution and computational resources. To increase the accuracy of the grid approximation, it is necessary to increase the spatial resolution of the grid, but doing so increases both the processing time and memory requirements by $O(n^3)$ for an environment in 3D. This can make it infeasible to simulate large environments accurately, especially when the fluid in those environments is very sparse. McAdams *et al.* [92] proposed a possible solution to this problem that utilized a multi-grid approach, simulating the fluid at different levels of resolution and then combing them together. While this can alleviate the extreme resolution vs. efficiency tradeoff somewhat, it has its own issues such as a difficult implementation, a challenge to parallelize on a GPU (critical to efficient fluid simulations), and many non-trivial failure cases. Instead, the main alternative to Eulerian fluid simulations, Lagrangian simulation, doesn't suffer from such an extreme tradeoff. Lagrangian fluid simulation represents the fluid as a set of particles, where each particle carries with it its own set of fluid properties, and a continuous vector field is approximated by interpolating between particles for any point in space [39]. The primary difference between Eulerian and Lagrangian methods is that the Eulerian method models the rate of fluid flow from one cell to another, whereas the Lagrangian method models the movement of individual "packets" of fluid. Bujack and Joy [17] phrased the difference well using an analogy:

> On one hand, the Eulerian specification describes the flow passing through a fixed spatial domain. It can be interpreted as the point of view of a stationary observer standing at a rivers bank watching the water flow by... On the other hand, the Lagrangian specification of a flow field describes the properties of a fixed fluid parcel as it travels through space. This specification can be imagined

as the point of view of an observer that sits in a boat and moves along with the flow of the river.

In this thesis we utilize the particle-based Lagrangian method because it is able to more efficiently represent sparse liquids in a large environment. The main algorithm that implements Lagrangian fluid dynamics is Smooth Particle Hydrodynamics (SPH) [39]. The key feature of this algorithm is that all the fluid properties in the Navier-Stokes equations (e.g., pressure, viscosity) are applied as physical forces to the particles, which are advected according to their momentum. SPH has been used extensively for astrophysical applications [37, 126] as well as modeling fluids for applications such as modeling lubrication between mechanical parts [76]. We use SPH for our experiments in chapter 7. However, SPH has one major drawback as it relates to liquids: it can't accurately model incompressible fluids such as water. There have been some proposals in the literature that offer various methods for modifying SPH to handle incompressible fluids [142, 13, 3]. Macklin *et al.* [90] proposed a different solution. They created Position Based Fluids (PBF), an extension of SPH that combines the approximate constraint solutions of position based dynamics [104] with the particle-based fluid representation of SPH. In PBF, the physical properties of fluids related to incompressibility (e.g., pressure) are treated as constraints rather than physical forces. During each update step, after the forces are applied and the particles advected, PBF uses an iterative, approximate solver to solve the incompressibility constraint. This way PBF is able to accurately and efficiently model incompressible fluids such as water. The authors released an implementation of their PBF algorithm in the commercially available physics simulator FleX [91]. We use PBF and FleX for our experiments in chapter 8.

Of course, incorporating fluid dynamics into our methodology adds complexity, so why not rather approximate the liquids as a set of small, rigid spheres? In fact, this was the methodology taken in [73, 74, 164, 165] when they needed to simulate liquids for their experiments. The drawback to this approach is that while the broad dynamics are the same (e.g., both will fall from an upturned container), the precise dynamics are not (e.g., rigid objects don't cohere as liquids do). For the work in this thesis, we are interested in enabling precise reasoning and control of liquids. Thus a rough approximation is not sufficient; we require a more precise model. So while using proper fluid dynamics does add complexity to our model, that complexity is a necessary condition of the precision we desire.

2.2.2 Learning Fluid Physics

A key contribution we make in this thesis is adding learning to the computational fluid dynamics methodology. There have been several attempts in the literature to combine learning and fluid dynamics. Elbrechter *et al.* [27] used a nearest neighbor and polynomial regression model to learn the correspondence between visual changes in a liquid's surface and its viscosity. In their experiments, the robot used a kinect depth camera to perceive the surface of the liquid, then proceeded to apply a pushing motion. Their model was successfully able to not only regress to known liquids, but also able to accurately predict the viscosity of unknown liquids as well. Guevara *et al.* [46, 45] worked on a similar problem. However, instead of regressing directly to viscosity as in [27], they used a fluid simulator to optimize the parameters of the liquid. The robot performed an action with the liquid (either pouring as in [46] or stirring as in [45]) and then would observe the changes caused by this action. The robot then internally simulated the same environment in a fluid simulator (they used Nvidia FleX [91]) and optimize the liquid parameters such as viscosity or cohesion until the simulator matched the real environment. Given this estimate for the liquid parameters, the robot could then successfully execute actions such as pouring without spilling. Relatedly, work by Mottaghi *et al.* [100, 99] has shown how physical scene models extracted from real data can be used to reason about physics, albeit rigid body physics in the case of [100, 99]. They utilized convolutional neural networks (CNNs) to convert an image into a description of a scene, and then applied Newtonian physics to understand what would happen in the future. Work on deformable objects, which share many properties with fluids (e.g., non-rigidity, dense state, etc.), by Schulman *et al.* [137] utilized physics models and a depth camera to track the surface of objects such as ropes, clothes, and sponges.

However the works in [27, 46, 45, 100, 99, 137] focused on utilizing physics for reasoning; other work has focused on learning the dynamics of fluids. Yamaguchi and Atkeson as part of [164, 162, 165] learned a dynamics model that used an abstract feature representation of the state of the fluid. Here they represented the state using the relative positions of the objects, the location and variance of the fluid stream, and the amount of fluid in the container vs. the amount of fluid spilled. As a result, the dyanmics learned were only only in the most broad, general terms over the abstract features. Kozek and Roska [71] were, to the best of the authors' knowledge, the first to directly model the full Navier-Stokes equations in a machine learning framework. The used cellular neural networks to compute dense fluid dynamics on a 2D grid, although the focus of [71] wasn't on learning. Long *et al.* [88] extended [71] by adding in a recurrent convolutional network to learn the motion of external forces (e.g., a moving wall), which then allowed them to predict the dense 2D future state of the fluid in the environment. Scott *et al.* [56] took a different approach, applying learning to the fluid dynamics themselves. They used multilayer perceptrons and support vector machines to learn to predict the height of waves in a bay along a coastline from detailed oceanographic data such as wave conditions, ocean-current nowcasts, and reported winds. Other work has also focused on learning detailed 2D dynamics. Schenck *et al.* [134][2] used fully convolutional networks to learn the 2D changes in a height map over granular media after scooping actions. While granular media and liquids are not the same, they share many similar properties, such as non-rigidity and dense state representations.

There has, however, been work focusing on learning fluid dynamics directly from data. Singh *et al.* [140] combined a standard fluid model with neural networks to learn air flow over various geometries, producing results superior to the fluid model alone. Tompson *et al.* [151] also combined a traditional fluid dynamics model with machine learning. They computed the advection of the fluid specified by the Navier-Stokes equations using a standard advection model and then used a convolutional neural network to solve the incompressability

[2]Several of the authors on this thesis were the primary contributors to this paper.

constraints. They represented the fluid using a dense 3D grid, which made it straightforward to interface with a convolutional network. However in this thesis we use a sparse particle set to represent fluid, which has better tradeoffs between computational resources and resolution. It is less obvious, though, how to perform learning using this representation, although there are some cases in the literature. Ladický *et al.* [77] learned fluid dynamics with a particle-based representation. They used regression forests to learn 3D fluid dynamics and applied the forest to each particle individually to compute it's dynamics. In order to facilitate particle-particle interaction, they pre-computed a set of features for each particle based on its relation to its neighbors. Because of this, the forest only had to consider the set of features for each particle and did not need to directly consider particle-particle relations. Mrowca *et al.* [102] also used a particle-based representation to compute physical interactions, albeit with deformable objects rather than fluids. They represented each object as a collection of particles, and used hierarchical graph convolutions to learn the deformation properties between particles belonging to the same object, allowing them to learn, for example, how much an object compresses when dropped from a given height. In chapter 8 we develop a similar method for computing particle-particle interactions using deep networks.

Part I

LEARNING-BASED METHODS

Chapter 3

LEARNING TO POUR: A FIRST LOOK AT LIQUID MANIPULATION

For the first experiment in this thesis we focus on a stereotypical task involving liquids: *pouring*. We task the robot with pouring a precise amount of liquid from a source container into a target container. To focus the investigation, we augment the robot's environment by placing a scale capable of real-time feedback under the target container. We simplify the perception of the robot in this chapter to allow us to get an initial sense of what a robot needs to be able to do to solve this task. We then apply what we learn in this chapter to investigations in future chapters.

To solve the *pouring* task in this chapter, we utilize reinforcement learning. Reinforcement learning is a well studied problem in machine learning [6]. Many researchers have successfully applied various reinforcement learning methods to different virtual domains [96, 15, 123, 93]. There have also been multiple studies applying reinforcement learning to other tasks on real robotic systems [141, 65]. Work by Konidaris *et al.* [66, 68, 67] and Niekum [106, 107] looked at how a robot can solve complicated tasks by chaining learned skills together. But these works represented learning of rather imprecise, though complex, goals (e.g., pressing a large button to open a door), which are not as suitable for learning precise control of liquids. Work by Deisenroth *et al.* [25, 24] on the PILCO framework and Levine *et al.* [83, 82] on Guided Policy Search (GPS) has shown how model-based reinforcement learning frameworks can be used for precise control tasks such as cart-pole or block insertion. We evaluated both for use in this chapter and empirically determined that the time-dependent locally-linear dynamics model used by GPS was more suited to the task at hand.

[0]The contents of this chapter were published in [129].

In this chapter, we apply reinforcement learning to the real-world task of pouring precise amounts of liquid from one container to another. Specifically, we utilize the reinforcement learning algorithm Guided Policy Search (GPS) [83] to learn a policy for pouring water. The goal of the robot is to pour a specific amount of liquid from a cup (source container) into a bowl (target container). The precise nature of the pouring task necessitates that the robot have at least a cursory understanding of the fluid dynamics involved, a highly non-trivial problem.

To solve this task, the robot first trains a dynamics model from a set of example trajectories. Then, given this dynamics model, it alternates between a single step of trajectory optimization and updating the weights of a neural network policy to match the optimized trajectories. Next, the robot rolls out the trained policy on the real system, using the resulting sensor data to retrain the dynamics model. This process repeats until the robot converges.

Our results show that a robot can successfully utilize this methodology to learn a policy enabling it to pour a precise amount of liquid. We varied the initial amount of water in the container, and for each initial amount, the robot attempted to pour a precise amount into the bowl. The robot converged after approximately 25 iterations. It was able to pour within 10g of the target for all initial conditions.

The rest of this chapter is organized as follows. The next section describes our experimental setup. Section 3.2 lays out the methodology we employed to solve this task in detail. Section 3.3 describes how we evaluated the robot on the pouring task. Section 3.4 details the results the robot was able to achieve. And finally section 3.5 concludes the chapter and describes what our investigation discovered.

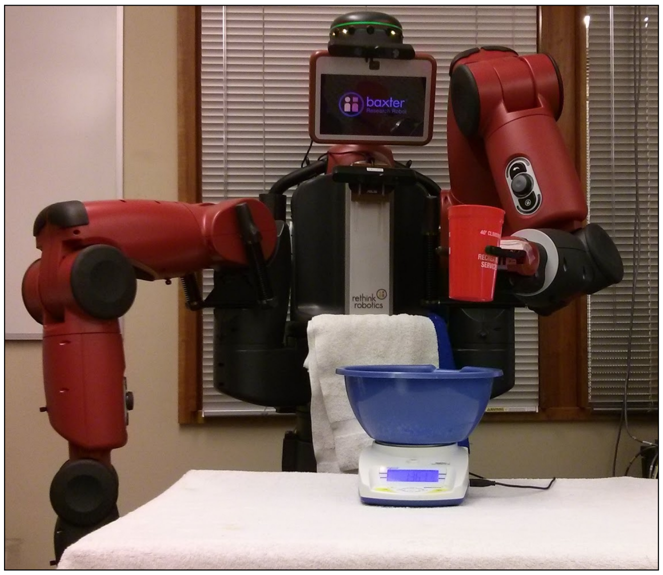


Figure 3.1: The robot used in these experiments. It is a Rethink Robotics Baxter Research Robot, equipped with two 7-DOF arms. Also shown is the experimental setup with the cup in the robot's gripper positioned above the table, with the bowl resting on top of the scale.

3.1 Experimental Setup

3.1.1 Robotic Platform

The robot used for the experiments in this chapter was the Rethink Robotics Baxter Research Robot, pictured in figure 3.1. It is an upper-torso humanoid robot with two 7-degree-of-freedom arms. Each arm is equipped with an electric parallel gripper. The motors on each joint can be controlled using position controls (via a built-in PID controller), velocity controls, or torque controls. We controlled the robot's arm using the velocity control mode. The joints are equipped with joint encoders and torque sensors[1]. We use the robot's left arm for all experiments in this chapter.

[1]We empirically determined that the torque sensors built into the robot's joints are not reliable and so did not use them here.

3.1.2 Experimental Environment

The robot was place in front of a small table. On the table was an Adam Equipment HCB 3000 Highland Portable Precision Balance scale. The scale had a maximum capacity of three kilograms and a resolution of one tenth of one gram. This scale was selected for its ability to provide real-time readings via USB cable to an attached computer. The scale has a built-in filter that introduces an approximately 0.5 to 1.0 second sensor delay in the measurements taken from the scale. On top of the scale was placed a medium sized plastic mixing bowl. A plastic cup was placed in the robot's left gripper. This configuration is shown in figure 3.1. The cup was pre-filled by the experimenters with a precise amount of water between two-hundred and four-hundred grams.

3.1.3 Data Collection

We ran 500 pouring trials. Each trial began with the cup already in the robot's gripper and pre-filled by the experimenter. The trial ended after exactly twenty-five seconds had passed, regardless of whether or not the robot had poured any liquid. During each trial, the robot's joint angles, joint velocities, and the scale value were recorded at a rate of two hertz. While the robot is equipped with many other sensors, the robot only used it's own joint angles, velocities, and the scale value to learn the pouring task.

3.1.4 State Space

In order to isolate the precise pouring task, we fixed the robot's wrist over the bowl on the table and gave it control only over it's last joint (the wrist joint), effectively letting it control only the angle of the cup. The robot used a four-dimensional state space. The first dimension was the angle of the robot's wrist in radians, shifted so that zero is when the cup is upright and π when the cup is inverted. The second dimension is the amount, in grams, remaining to pour until the robot reaches the pouring target. In this way, the target pouring amount is implicitly built into the state space. The third dimension is the change in the second dimension from the last timestep to the current one. And finally, the fourth dimension is the amount of water in the cup. This amount is initialized to a specific, known amount, and then updated throughout each trial by subtracting the change in the scale value.

3.2 Methodology

3.2.1 Problem Definition

Let $\mathcal{X} \in \mathbb{R}^d$ be the d dimensional state space the robot operates in, and let $\mathcal{X}_{init} \subseteq \mathcal{X}$ be the set of valid starting states. The goal of the robot is to learn a policy $\pi(\mathbf{x}_t; \theta) \rightarrow \mathbf{u}_t$ that minimizes a cost function l, where $\mathbf{x}_t$ is the state at time t, $\mathbf{u}_t$ is the robot's control signal at time t, and θ is the learned policy parameters. The robot must learn a policy that, from any initial state $x_1 \in \mathcal{X}_{init}$, generates a trajectory $(\mathbf{x}_1, \mathbf{u}_1), ..., (\mathbf{x}_T, \mathbf{u}_T), (\mathbf{x}_{T+1})$ that minimizes $\sum_{i=1}^{T} [l(\mathbf{x}_i, \mathbf{u}_i)] + l(\mathbf{x}_{T+1})$ over a fixed horizon T.

3.2.2 Algorithm Overview

We use a modified version of Guided Policy Search [83] to train policy parameters θ. The robot is given N initial example trajectories $\tau = \{(\mathbf{x}_1, \mathbf{u}_1), ..., (\mathbf{x}_T, \mathbf{u}_T), (\mathbf{x}_{T+1})\}^N$, where $\mathbf{x}_t$ is the state of the robot at time t and $\mathbf{u}_t$ is the control applied at time t. Next the robot trains a time-based dynamics model $\hat{f}_t(\mathbf{x}_t, \mathbf{u}_t) \rightarrow \mathbf{x}_{t+1}$. Using this model, the robot alternates between optimizing the trajectories τ using an iLQG [146] backward pass and optimizing the policy parameters θ to match the updated trajectories. Next the robot uses the policy $\pi(\mathbf{x}_t; \theta) \rightarrow \mathbf{u}_t$ to rollout from each starting state $\mathbf{x}_1$ in each trajectory τ_i. Finally, the robot retrains the dynamics model $\hat{f}_t$ and repeats this process until convergence.

3.2.3 Learning the Dynamics Model

We use a very similar learned dynamics model to [82]. The robot must learn the function $\hat{f}(\mathbf{x}_t, \mathbf{u}_t) \rightarrow \mathbf{x}_{t+1}$ mapping the state $\mathbf{x}_t$ and control $\mathbf{u}_t$ at the current timestep to the next state $\mathbf{x}_{t+1}$. The dynamics model is a time-based, locally linear model with a Gaussian mixture model over the prior. Since the model is local, the robot learns a separate model for each of the N example trajectories. The rest of this section will describe the training process for one model.

Given a training set $\tilde{\tau} = \{(\mathbf{x}_1, \mathbf{u}_1), ..., (\mathbf{x}_T, \mathbf{u}_T), (\mathbf{x}_{T+1})\}^M$, where M is the number of iterations so far, the robot learns a separate linear function $\hat{f}_t(\mathbf{x}_t, \mathbf{u}_t) \rightarrow \mathbf{x}_{t+1}$ for all timesteps t. For applications on real robots, though, the number of training trajectories M can often be too small compared to the dimensionality of the state space to fit a linear model at each timestep, so instead the robot learns an equivalent model and uses shared dynamics between timesteps to compensate for the small amount of training data at each timestep.

At each timestep, the robot fits a multivariate Gaussian $\mathcal{N}(\mu_t, \Sigma_t)$ over the concatenated vectors $\langle \mathbf{x}_t, \mathbf{u}_t, \mathbf{x}_{t+1} \rangle$ which we write as $\langle \mathbf{x}, \mathbf{u}, \mathbf{x}' \rangle$ for simplicity. To predict the next state after timestep t given $\mathbf{x}$ and $\mathbf{u}$, the robot can simply condition the normal distribution on $\mathbf{x}$ and $\mathbf{u}$ and solve for $\mathbf{x}'$ as follows:

$$\mathbf{x}' = \mu_{\mathbf{x}'} + \Sigma_{\mathbf{x}'\langle \mathbf{x},\mathbf{u}\rangle} \Sigma^{-1}_{\langle \mathbf{x},\mathbf{u}\rangle\langle \mathbf{x},\mathbf{u}\rangle} \left(\langle \mathbf{x}, \mathbf{u} \rangle - \mu_{\langle \mathbf{x},\mathbf{u}\rangle} \right)$$

where μ_a is the components of μ who's elements pertain to a, and Σ_{ab} is the covariance between a and b extracted from Σ. Note that we only use the mean of the conditional distribution over $\mathbf{x}'$ in this paper.

Estimating the Model Parameters

Let $\bar{\mu}$ and $\bar{\Sigma}$ be the empirical mean and covariance respectively for all $\langle \mathbf{x}_t, \mathbf{u}_t, \mathbf{x}_{t+1} \rangle$ for all t. The robot estimates an inverse-Wishart prior over the parameters of the T Gaussian distributions $\mathcal{N}(\mu_t, \Sigma_t)$. The prior has parameters $\mathbf{\Phi}$, μ_0, m, and n. The parameters μ_t and Σ_t for the distribution at time t are estimated as follows:

$$\mu_t = \frac{m\mu_0 + M\hat{\mu}_t}{m + M} \qquad \Sigma_t = \frac{\mathbf{\Phi} + M\hat{\Sigma}_t + \frac{Mm}{n+m}(\hat{\mu}_t - \mu_0)(\hat{\mu}_t - \mu_0)^T}{M + n_0}$$

where $\hat{\mu}_t$ and $\hat{\Sigma}_t$ are the empirical mean and covariance respectively of the set $\{\langle \mathbf{x}_t, \mathbf{u}_t, \mathbf{x}_{t+1} \rangle\}^M$.

The prior parameters $\mathbf{\Phi}$, μ_0, m, and n are estimated as

$$\mathbf{\Phi} = n_0\bar{\Sigma} \qquad \mu_0 = \bar{\mu} \qquad m = n_0 = 1.$$

Approximating Non-Linearity With a Gaussian Mixture Model

While the above method for estimating the Gaussian parameters μ_t and Σ_t effectively reduces the number of training data points required at each time point t, the inverse-Wishart prior enforces a global linearity assumption on the dynamics model, as opposed to a local linearity assumption, which can make it difficult for the robot to operate in highly non-linear environments.

To handle this, the robot fits a Gaussian mixture model [94] over the set $\{\langle \mathbf{x}_t, \mathbf{u}_t, \mathbf{x}_{t+1}\rangle\}^M_{t=1,\ldots,T}$. Then, for each $t \in [1, T]$, the robot estimates the prior parameters $\mathbf{\Phi}$ and μ_0 (n_0 and m remain constant at 1) as

$$\Phi = \frac{\sum\limits_i p(\{\langle \mathbf{x}_t, \mathbf{u}_t, \mathbf{x}_{t+1}\rangle\}^M | \bar{\mu}_i, \bar{\Sigma}_i)\bar{\Sigma}_i}{\sum\limits_i p(\{\langle \mathbf{x}_t, \mathbf{u}_t, \mathbf{x}_{t+1}\rangle\}^M | \bar{\mu}_i, \bar{\Sigma}_i)}$$

$$\mu_0 = \frac{\sum\limits_i p(\{\langle \mathbf{x}_t, \mathbf{u}_t, \mathbf{x}_{t+1}\rangle\}^M | \bar{\mu}_i, \bar{\Sigma}_i)\bar{\mu}_i}{\sum\limits_i p(\{\langle \mathbf{x}_t, \mathbf{u}_t, \mathbf{x}_{t+1}\rangle\}^M | \bar{\mu}_i, \bar{\Sigma}_i)}$$

where $\bar{\mu}_i$ and $\bar{\Sigma}_i$ are the mean and covariance of the ith mixing element. Essentially, Φ and μ_0 are the weighted average of each mixing element. In this paper, we used the standard implementation of Gaussian mixture models built into the Matlab Statistics and Machine Learning Toolbox [1].

3.2.4 Trajectory Optimization

Given a trained dynamics model $\hat{f}$ and set of example trajectories τ, the robot must optimize those trajectories with respect to a given cost function l. However, if the robot uses l directly during trajectory optimization, the policy π may be unable to approximate the example trajectories. Instead the robot uses the modified cost function

$$l^*(\mathbf{x}, \mathbf{u}, \pi) = l(\mathbf{x}, \mathbf{u}) + \lambda \|\mathbf{u} - \pi(\mathbf{x})\|^2$$

where $\|x\|$ is the L2-norm and λ is the weight given to the second term of the equation. Informally, the second term of l^* enforces that whatever trajectory the optimizer finds, it should stay close to the policy.

iLQG Backward Pass

For each trajectory $\tau_i \in \tau$, the robot performs an iLQG backward pass to optimize the trajectory. It splits the combined optimization problem over $\mathbf{u}_1, ..., \mathbf{u}_T$ into individual optimizations for each $\mathbf{u}_t$ by optimizing the Q-function backwards in time, starting at time T. The Q-function is given by

$$Q(\delta\mathbf{x}_t, \delta\mathbf{u}_t) = l^*(\mathbf{x}_t + \delta\mathbf{x}_t, \mathbf{u}_t + \delta\mathbf{u}_t) + V_{t+1}(\hat{f}(\mathbf{x}_t + \delta\mathbf{x}_t, \mathbf{u}_t + \delta\mathbf{u}_t))$$

where $\delta\mathbf{x}_t$ and $\delta\mathbf{u}_t$ are the updates to apply to $\mathbf{x}_t$ and $\mathbf{u}_t$ respectively, and V_{t+1} is given by

$$V_{t+1}(\mathbf{x}) = \min_{\mathbf{u}} \left[l^*(\mathbf{x}, \mathbf{u}) + V_{t+2}(\hat{f}(\mathbf{x}, \mathbf{u})) \right].$$

The robot minimizes Q with respect to $\delta\mathbf{u}_t$ by taking the first and second derivatives of Q. V is intractable to differentiate directly, so the robot approximates V_{t+1} by substituting the next timestep's result, $Q(\delta\mathbf{x}_{t+1}, \delta\mathbf{u}_{t+1})$, in place of V_{t+1}. Thus it is necessary to work backwards through the trajectory so that $Q(\delta\mathbf{x}_{t+1}, \delta\mathbf{u}_{t+1})$ is already computed when computing $Q(\delta\mathbf{x}_t, \delta\mathbf{u}_t)$.

For further details of the iLQG algorithm, please refer to [146].

3.2.5 Policy Learning

Given the set of example trajectories $\tau = \{(\mathbf{x}_1, \mathbf{u}_1), ..., (\mathbf{x}_T, \mathbf{u}_T), (\mathbf{x}_{T+1})\}^N$, fitting the policy parameters θ can be framed as a simple regression problem, i.e., train the parameters to map every $\mathbf{x}_t \rightarrow \mathbf{u}_t$. However, if N and T are small (e.g., 10 and 50, respectively), then that can leave relatively few training points for high-dimensional state spaces, which can make it difficult for the policy to smoothly fit a function over the uncovered areas of the space.

To solve this, the robot utilizes the gains matrices L_t generated during the iLQG backward pass to generate more training data points. The matrix L_t is used as follows

$$\mathbf{u}_t = \hat{\mathbf{u}}_t + L_t \left(\mathbf{x}_t - \hat{\mathbf{x}}_t\right)$$

where $\hat{\mathbf{x}}_t$ and $\hat{\mathbf{u}}_t$ are the open-loop trajectory found by iLQG at time t, and $\mathbf{x}_t$ and $\mathbf{u}_t$ are the actual state and controls when rolling out the trajectory on the real system. Intuitively, L_t computes how much to alter the open-loop control $\hat{\mathbf{u}}_t$ based on the difference between the open-loop state $\hat{\mathbf{x}}_t$ and the actual state $\mathbf{x}_t$. Thus, L_t makes the open-loop trajectory into a local policy around that trajectory.

To generate more data points, the robot draws many samples from a small Gaussian around each $\mathbf{x}_t \in \tau$, and then uses the corresponding gains matrix L_t to generate the controls for each sampled point. Adding these sampled data points to the original training data points, the robot can then formulate learning the policy parameters θ as a standard regression problem. For the experiments in this chapter, the robot used a neural network to learn the policy.

Note that it is critical to the success of the robot to tightly interleave the trajectory optimization and policy learning iterations. That is, the robot does exactly one iLQG backward pass on each example trajectory, followed by updating the policy parameters θ to better fit the example trajectories. The robot repeats this inner loop multiple times before rolling out the policy in the real environment, and then returning to the inner loop. If the trajectory optimization and policy learning were not tightly interleaved, then the trajectory optimization could easily optimize the example trajectories in a way that would make it very difficult or impossible for the policy to learn. Instead, interleaving them in this manner keeps the policy "close" to the trajectory optimizer, and the policy deviation term in the cost function prevents the trajectory optimizer from moving the trajectories too far from what the policy can learn.

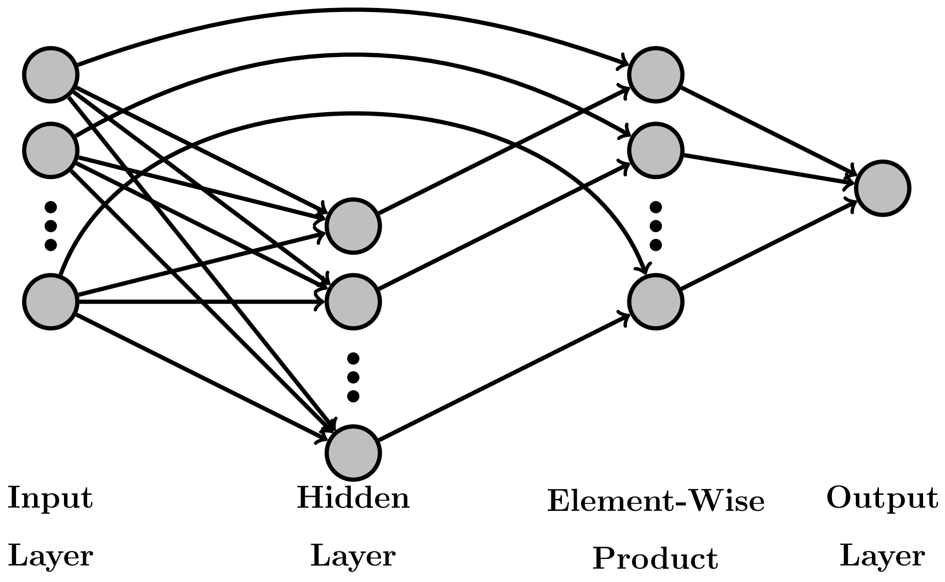


Figure 3.2: The neural network used to learn the policy. The input layer is fed into a fully-connected hidden layer, and then the output of the hidden layer is element-wise multiplied by the input layer, which is finally fed into the output layer.

Training the Neural Network

The neural network layout used by the robot is shown in figure 3.2. The input $\mathbf{x}_t$ is fed into a hidden layer with $|\mathbf{x}_t|$ hidden units. The output of each is the linear combination of its input fed through a rectified linear function, i.e.,

$$h_i(\mathbf{x}_t) = max\left(\mathbf{x}_t \bullet \mathbf{w}_i, 0\right)$$

where $\mathbf{w}_i$ is the set of weights for node i. Next, the output of the hidden layer is multiplied element-wise with the input. Finally, the result of the element-wise product is combined linearly into the output of the network.

This neural network was implemented using the Caffe deep learning framework [58]. We used the built-in backpropagation to fit the weights of the network to the training data.

3.2.6 Handling Delay

In the experiments we run in this chapter, there is a non-trivial amount of sensor delay that complicates the problem. In section 3.1.2, while describing the experimental environment, we stated that the real-time scale underneath the target container has a delay of 0.5s to 1s between when the weight changes and when it sends that change to the robot. Furthermore, there is also delay between when the robot changes the pose of the source container and when the liquid falls and lands in the target. Combined, these two mean that there will be a delay between when the robot takes an action and when that action affects the state. In order to learn in an environment with delay, the robot augments it's reasoning about states to include the previous n states. Thus, when learning the dynamics model, the function maps the previous n states and controls to the next state, i.e., $\hat{f}(\mathbf{x}_{t-n}, \mathbf{u}_{t-n}, ..., \mathbf{x}_t, \mathbf{u}_t) \rightarrow \mathbf{x}_{t+1}$. Additionally, when learning the policy, it takes into account the previous n states as well, i.e., $\pi(\mathbf{x}_{t-n}, ..., \mathbf{x}_t; \theta) \rightarrow \mathbf{u}_t$. So long as the previous n states cover the length of the delay, reasoning about them allows the robot to combine both delayed and non-delayed sensor readings to predict the next state or control.

3.3 Evaluation

We evaluated the robot on the pouring task. The robot's arm was fixed over the bowl, and a cup was placed in its gripper. It was given control over its wrist joint so that it could control the angle of the cup. The robot controlled the angle by setting the angular velocity of its wrist joint. The cup was initialized upright before each trial and prefilled by the experimenter. The goal of the robot was to pour a specific amount of water into the bowl, and to be as accurate as possible.

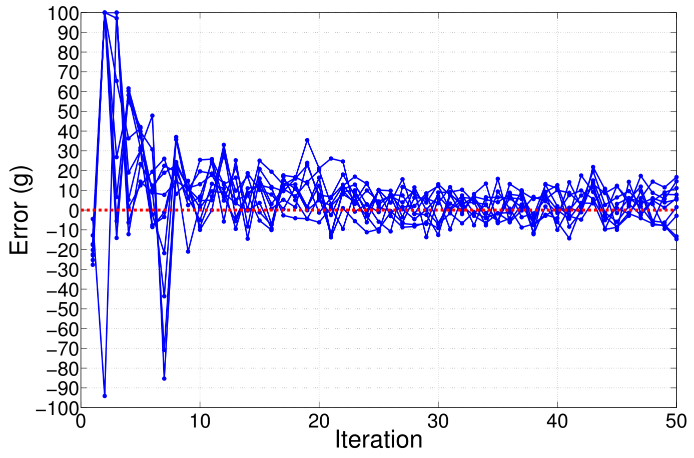
We set the number of example trajectories N to 10, and varied the initial amount of water in the cup for each trajectory uniformly between 200 and 400 grams, and we fixed the pouring target at 100 grams. The trajectories were initialized using a standard PID controller. We fixed n, the number of previous states to include, at 4 (i.e., the current state plus the 3 previous). The horizon for every trajectory was fixed to 50 timesteps, which, given a 2 hertz sampling rate, meant each trajectory lasted exactly 25 seconds. We chose to sample at 2 hertz to balance between having n high enough to cover the entire length of the sensor delay yet low enough to not cause the augmented state space to be too high-dimensional.

At the start of each iteration, the robot trained the dynamics model. Next, the robot alternated between one step of trajectory optimization and fitting the policy 10 times each. After finishing the inner loop of the algorithm, the robot rolled out each of the 10 trajectories from their starting states on the real system using the learned policy. Finally, the robot updated each example trajectory with the results of the real rollout, and then began the next iteration.

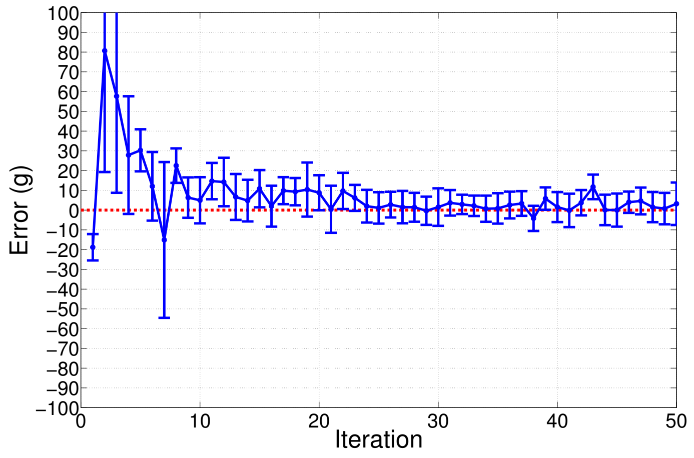
3.4 Results

The results are shown in figure 3.3. The error is reported in the number of grams of deviation the robot was from the target, that is, after a rollout, the difference between the number of grams of water in the bowl and the desired number, with values closer to 0 meaning better performance. Figure 3.3a shows the error for each example trajectory after each iteration. Figure 3.3b shows the mean and standard deviation of the example trajectories shown in figure 3.3a.

From the graphs, it is clear that the robot converged after approximately 25 iterations. Looking closely at figure 3.3a, it is apparent that iteration 33 was the first iteration where the robot was able to pour within 10 grams of the target for all trajectories. Furthermore, figure 3.3b shows that after iteration 23, the standard deviation of the robot's error falls below 10 grams. In our experience, an accuracy of ± 10 grams is approximately what can be expected of a human performing the same task. Thus, the graphs in figure 3.3 show that the robot, after approximately 25 iterations, was able to converge to human-level performance.



(a) Error for each of the example trajectories



(b) Mean and standard deviation of the error

Figure 3.3: The error in grams after each iteration. The error is how far the robot was from the pouring target at the end of the rollout.

3.5 Discussion

In this chapter, we used Guided Policy Search to train a policy on a real robot to solve a task with non-trivial sensor delay. Specifically, the robot learned a policy for the pouring task. The goal of the robot was to pour a precise amount of water. It did this by iteratively pouring with its current policy, training a dynamics model, and then updating the policy using trajectory optimization. The robot was able to pour within 10 grams of the target (100 grams) after 33 iterations.

We showed that the robot was able to reach human-levels of performance on the pouring task. While this may not be high enough for tasks such as high precision manufacturing, it is sufficient for many household tasks such as cooking. Furthermore, we showed that, using a generic robotic platform, a robot can successfully learn to manipulate fluids. This is important for work in later chapters. Here we showed how that if the robot could know the rate of change of liquid from the cup to the bowl, then it can successfully pour precise amounts. In the next chapter we examine how a robot can eschew the real-time scale and instead use a generic color camera to perceive the liquids. After that, in chapter 5 we come back to the problem of pouring precise amounts of liquids, albeit using a camera instead of a scale. What we learned in this chapter showed us how a robot can pour specific amounts of liquid, and we apply that in the next two chapters.

3.5.1 Practical Takeaways

In addition to the insights discussed above, there are several more technical insights we found while performing the research in this chapter. Most of the motivation for our choices in methodology was due to the sensor delay introduced by the real-time scale. The scale used a proprietary algorithm built-in to the microcontroller in the scale itself to smooth the output values. This algorithm was not consistent and caused a delay of approximately 0.5 to 1 seconds between when the mass on the scale changed and when it reported the change. Additionally, because the algorithm was proprietary, we could not counteract this varied delay in our own code. We initially attempted to use the PD controller (the controller we used to initialize the example trajectories for GPS) to solve the task directly, without significant learning. However, due to the variable delay, the PD controller's performance was inconsistent, sometimes performing well and sometimes overpouring by a significant amount due to a large sensor delay. The only way to guarantee good performance was to make the controller pour incredibly slowly, a not very practical approach.

Due to this unknown, variable delay, we determined that a learning approach was necessary. We initially attempted to use PILCO [24], a framework for learning control policies from data. Similar to GPS, PILCO uses the example trajectories to train a dynamics model. However, PILCO uses Gaussian processes rather than a time-varying locally-linear model to learn the dynamics, and uses the direct policy gradients method to update the parameters of the policy. We spent much research time (several months) attempting to teach the robot to pour using PILCO to no avail. The Gaussian processes were unable to accurately model the dynamics, even with the inclusion of several prior states. We switched to the time-varying locally-linear models used by GPS, which were able to model the dynamics, however only locally. Because of this, the direct policy gradient method employed by PILCO was not applicable, so we switched to GPS. By using GPS, we were able to successfully teach the robot to pour precise amounts of liquid.

Chapter 4

LEARNING TO SEE: TEACHING ROBOTS TO PERCEIVE LIQUIDS

In the last chapter we looked at how the robot could control liquids in a precise pouring task. There, the robot used a real-time scale to sense the rate of transfer. However, this kind of sensor will not always be available and it may not always be possible to augment the environment with it. One alternative to a scale are the force-torque sensors in a robot's arm. While these may be useful in expensive industrial robots, in inexpensive modern robots, the kind most likely to appear in unstructured homes due to their lower cost, force-torque sensors are often noisy and unreliable, making them unsuitable for precision control. Another alternative is to use a commercially available depth sensor such as a Microsoft Kinect, which has been used for rigid object localization [135]. However, due to their translucent nature, liquids do not reliably appear on depth cameras. Instead, we examine perceiving liquids using a more common and inexpensive general-purpose sensor, a color camera.

Specifically, we examine the problem of *perceiving* liquids. That is, we ask the question *Where in the raw visual data stream is liquid?* To solve this problem, we take advantage of recent advances in the field of deep learning. This approach has been extremely successful in various areas of computer vision, including classification [72], semantic labeling [29], and pose regression [38], and it enabled computers to successfully play Atari games from raw image data [47] and train end-to-end policies on robots [82]. The ability of deep networks to process and make sense of raw visual data makes them a good fit for the task of perceiving liquids.

In this chapter, we focus on the task of pouring as our exemplar task for learning

[0]The contents of this chapter were published in [130] and [133].

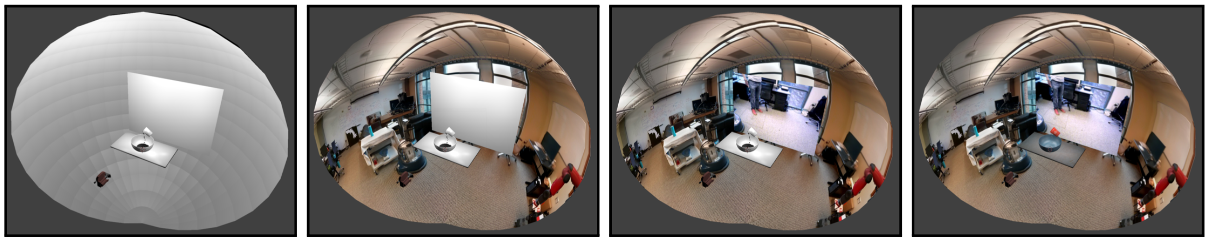
about liquids. While researchers have already worked on robotic pouring tasks, previous techniques made simplifying assumptions, such as replacing water by an easily visible granular medium [165], restricting the setting such that no perceptual feedback is necessary [79, 108, 144, 18], requiring highly accurate force sensors [127], detecting moving liquid in front of a relatively static background [163], or dealing with simulated liquids only [73, 74]. Here, we show how fully-convolutional deep networks (FCNs) can be trained to robustly perceive liquids and how they can be modified to perform better at generalization. To collect the large amounts of data necessary to train these deep networks, we utilize a realistic liquid simulator to generate a simulated dataset and a thermal camera to automatically label water pixels in a dataset collected on the real robot.

Our results show that the methodology we propose in this paper is able to *perceive* liquids quite well. Specifically, they show that recurrent networks are well-suited to this task, as they are able to integrate information over time in a useful manner. We also show that, with the right format of the input image, our neural networks can generalize to new data with objects that are not included in the training set. These results strongly suggest that our deep learning approach is useful in a robotics context, which we demonstrate in the following chapter with a closed-loop water pouring experiment.

The rest of this chapter is laid out as follows. The next section describes the task we investigate in this paper. The sections after that describe how we generate our simulated dataset and performed the pouring trials on our robot, followed by a discussion of our learning methodology. We then describe how we evaluate our networks and present experimental results. And finally, the last section concludes the chapter with a discussion of the results.

4.1 Task Overview

In this chapter we investigate the task of *perceiving* liquids. We evaluate the robot on the *perception* task of *detection.* We define *detection* to be determining what in the raw sensory data is liquid, and what is not liquid. Specifically, given an image, the robot must produce pixel-wise labels *liquid* for pixels containing liquid and *not-liquid* for pixels not containing



(a) *Untextured* (b) *Background* (c) *Background+Video* (d) *Fully Textured*

Figure 4.1: The scene used to simulate pouring liquids. The background sphere is cut-away to show its interior. From left to right: The scene shown without any texture or materials; The background image sphere texture added; The video on the plane added in addition to the background texture; and The scene fully textured with all materials.

liquid. For this chapter, we focus on the task of pouring as it is a common task involving liquids and presents a challenging perception problem. We also consider images in the context of sequences, that is, we don't consider images as static scenes in isolation but rather as a part of a series, and we allow the robot to utilize that in solving the task.

We evaluate our neural networks on the task of *detection* in both simulation and on data collected on a real robot. For the simulated dataset, we generated a large amount of pouring sequences using a realistic liquid simulator. As it is simple to get the ground truth state from the simulator, we can easily evaluate our methodology on the simulated data. For evaluations using real-world data, we carried out a series of pouring trials on our robot. We use a thermal camera in combination with heated water to acquire the ground truth pixel labels.

4.2 Simulated Data Set

We use a simulated to perform some of the evaluations of our methodology. The dataset contains 10,122 pouring sequences that are 15 seconds long each, for a total of 4,554,900 images. Each sequence was generated using the 3D-modeling program Blender [11] and the library El'Beem for liquid simulation, which is based on the lattice-Boltzmann method for efficient, physically accurate liquid simulations [69].

We divide the data generation into two steps: liquid simulation and rendering. Liquid simulation involves computing the trajectory of the mesh of the liquid over the course of the pour. Rendering is converting the state of the simulation at each point in time into color images. Liquid simulation is much more computationally intensive than rendering[1], so by splitting the data generation process into these two steps, we can simulate the trajectory of the liquid and then re-render it multiple times with different render settings (e.g., camera pose) to quickly generate a large amount of data. We describe these two steps in the following sections.

4.2.1 Liquid Simulation

The simulation environment was set up as follows. A 3D model of the target container was placed on a flat plane parallel to the ground, i.e., the "table." Above the target container and slightly to the side we placed the source container. This setup is shown in Figure 4.1a. The source container is pre-filled with a specific amount of liquid. The source then rotates about the y-axis following a fixed trajectory such that the lip of the container turns down into the target container. The trajectory of the liquid is computed at each timestep as the source container rotates. Each simulation lasted exactly 15 seconds, or 450 frames at 30 frames per second.

For each simulation, we systematically vary 4 variables:

- *Source Container* - *cup*, *bottle*, or *mug*
- *Target Container* - *bowl*, *dog dish*, or *fruit bowl*
- *Fill Amount* - 30%, 60%, or 90%

[1]Generating one 15 second sequence takes about 7.5 hours to simulate the liquid and an additional 0.5 hours to render it on our Intel Core i7 CPUs.

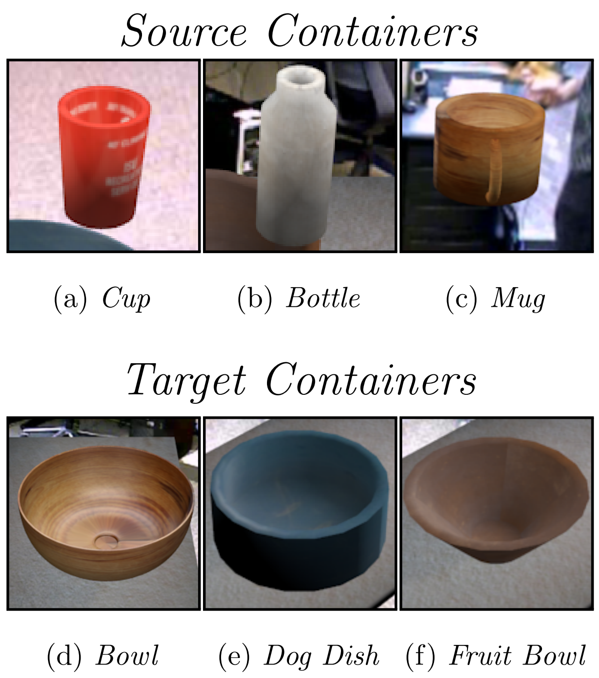


Figure 4.2: The objects used to generate the simulated dataset. The first row are the three source containers. The last row are the 3 target containers. The objects are each shown here with 1 of their possible 7 textures.

- *Trajectory* - partial, hold, or dump

The 3 source containers we used are shown in Figures 4.2a, 4.2b, and 4.2c, and the 3 target containers we used are shown in Figures 4.2d, 4.2e, and 4.2f. Each source container was filled either 30%, 60%, or 90% full at the start of each simulation. The source was rotated along one of three trajectories: It was rotated until it was slightly past parallel with the table, held for 2 seconds, then rotated back to upright (partial); It was rotated until it was slightly past parallel with the table, where it stayed for the remainder of the simulation (hold); or It was rotated quickly until it was pointing nearly vertically down into the target container, remaining there until the simulation finished (dump). The result was 81 liquid simulations (3 sources $\times$ 3 targets $\times$ 3 fill amounts $\times$ 3 trajectories).

4.2.2 Rendering

To generate rendered pouring sequences, we randomly select a simulation and render parameters[2]. We place the camera in the scene so that it is pointing directly at the table top where the target and source containers are. In order to approximate realistic reflections on the liquid's surface, we enclose the scene in a sphere with a photo sphere taken in our lab set as the texture (shown in Figure 4.1b). Next we place a video of activity in our lab behind the table opposite the camera (shown in Figure 4.1c). We took videos such that they approximately match the location in the image on the background sphere behind the video plane. We randomly select a texture for the source and target containers, and we render the liquid as 100% transparent (but including reflections and refractions). We also vary the reflectivity of the liquid as well as its index of refraction to simulate slight variations in the liquid type. Figure 4.1d shows the full scene with textures, video, and background sphere.

We randomly select from the following N parameters for each rendered sequence:

- *Source Texture* - 7 preset textures
- *Target Texture* - 7 preset textures
- *Activity Video* - 8 videos
- *Liquid Reflectivity* - normal or none
- *Liquid Index-of-Refraction* - air-like, low-water, or normal-water
- *Camera Azimuth* - 8 azimuths
- *Camera Height* - high or low
- *Camera Distance* - close, medium, or far

[2]The number of parameters makes it infeasible to evaluate every possible combination.

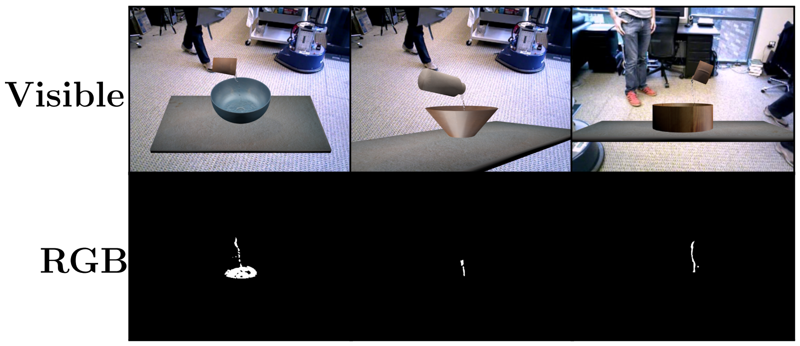


Figure 4.3: Examples of frames from the simulated dataset. The top row is the raw RGB images generated by the renderer; the bottom row shows the ground truth liquid location for visible liquid.

There are 48 total camera viewpoints. The camera azimuth is randomly selected from 1 of 8 possibilities spaced evenly around the table. The height of the camera is selected such that it is either looking down into the target container at a 45 degree angle (high, left column in Figure 4.3) or it is level with the table looking directly at the side of the target (low, right column in Figure 4.3). The camera is placed either close to the table, far from the table, or in between. The output of the rendering process is a series of color images, one for each frame of the sequence.

4.2.3 Generating the Ground Truth

We generate the ground truth for each image in each rendered sequence as follows. We set the liquid mesh to render as a solid color irrespective of lighting. Then we make all other objects in the scene solid black irrespective of lighting. This makes it simple to distinguish between which pixels contain liquid and which do not. Several examples are shown in figure 4.3) with the rendered RGB image on the top and the corresponding pixel labels on the bottom.

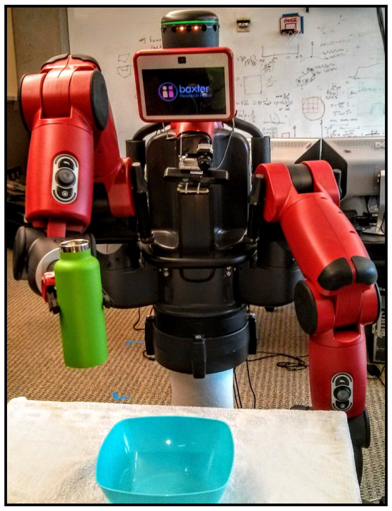


Figure 4.4: The robot used in the experiments in this chapter. It is shown here in front of a table, holding the *bottle* in its right gripper, with the *fruit bowl* placed on the table.

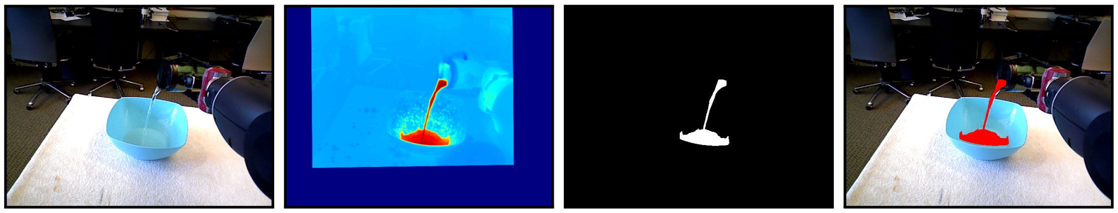
4.3 Robot Data Set

4.3.1 Robot

The robot used to collect the dataset is shown in Figure 4.4. It is a Rethink Robotics Baxter Research Robot, an upper-torso humanoid robot with 2 7-dof arms, each with a parallel gripper. The robot is placed in front of a table with a towel laid over it to absorb spilled water. The robot is controlled via joint velocity commands. In the experiments in this chapter, the robot uses only one of its arms at a time. The arm is fixed above the target container and the robot controls the joint velocity of its last joint, i.e., the rotational angle of its wrist.

4.3.2 Sensors

The robot is equipped with a pair of cameras mounted to its front immediately below its screen. The first camera is an Asus Xtion RGBD camera, capable of providing both color and depth images at 640×480 resolution and 30 Hz. The second camera is an Infrared Cameras



(a) *RGB* (b) *Thermal* (c) *Threshold* (d) *Overlay*

Figure 4.5: An example of obtaining the ground truth liquid labels from the thermal camera. From left to right: The color image from the RGBD camera; The thermal image from the thermal camera transformed to the color pixel space; The result after thresholding the values in the thermal image; and An overlay of the liquid pixels onto the color image.

Inc. 8640P Thermographic camera, capable of providing thermal images at 640×512 resolution and 30 Hz. The thermal camera is mounted immediately above the RGBD camera's color sensor, and is angled such that the two cameras view the same scene from largely similar perspectives. The Baxter robot is also equipped with joint-torque sensors, however the signal from these sensors is too unreliable and so we did not use them in these experiments.

Calibration of the Thermal Camera

For our experiments, we use the thermal camera in combination with heated water to acquire the ground truth pixel labels for the liquid. To do this, we must calibrate the thermal and RGBD cameras to each other. In order to calibrate the cameras, we must know the correspondence between pixels in each image. To get this correspondence, we use a checkerboard pattern printed on poster paper attached to an aluminum sheet. We then mount a bright light to the robot's torso and shine that light on the checkerboard pattern while ensuring it is visible in both cameras. The bright light is absorbed at differing rates by the light and dark squares of the pattern, resulting in a checkerboard pattern that is visible in the thermal camera[3].

We then use OpenCV's `findChessboardCorners` function to find the corners of the pattern in each image, resulting in a set of correspondence points P^{therm} and P^{RGB}. We compute the affine transform T between the two sets using singular-value decomposition. Thus to find the corresponding pixel from the thermal image to the RGB image, simply multiply as follows

$$Tp^{therm} = p^{RGB}$$

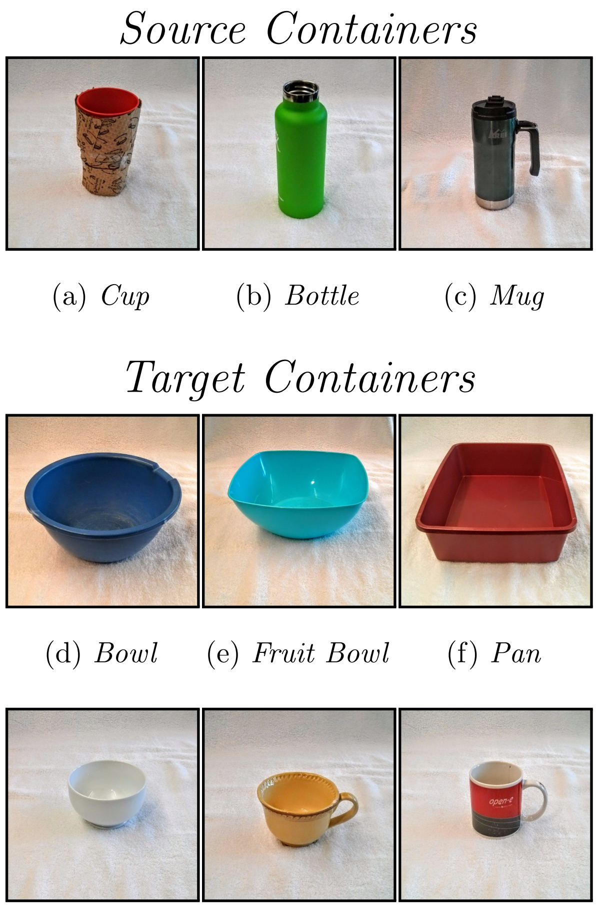
where p^{therm} is the xy coordinates of a pixel in the thermal image, and p^{RGB} are its corresponding xy coordinates in the RGB image.

It should be noted that T is only an affine transformation in pixel space, not a full registration between the two images. That is, T is only valid for pixels at the specific depth for which it was calibrated, and for pixels at different depths, Tp^{therm} will not correspond to the same object in the RGB image as p^{therm} in the thermal image. While methods do exist to compute a full registration between RGB and thermal images [116], they tend to be noisy and unreliable. For our purposes, since the liquids are always a constant depth from the camera, we opted to use this affine transform instead, which is both faster and more reliable, resulting in better ground truth pixel labels. While our RGBD camera does provide depth values at each pixel, the liquid does not appear in the depth readings and thus we could not use them to compute the full registration. Figure 4.5 shows an example of the correspondence between the thermal image and the RGB image.

4.3.3 Objects

For the robot dataset, we used two sets of objects: *source containers* and *target containers*. We used 3 different source containers, the *cup*, the *bottle*, and the *mug*, shown in 4.6a, 4.6b, and 4.6c. The *bottle* and *mug* were both thermally insulated, and we wrapped the *cup* in insulators. This was done so that the robot could use the same source container from trial

[3]Albeit inverted as the black squares absorb more light than the white, thus appearing brighter in the thermal image. However we only care about the corners of the pattern, which are the same.



(a) *Cup* (b) *Bottle* (c) *Mug*

(d) *Bowl* (e) *Fruit Bowl* (f) *Pan*

(g) *Small Bowl* (h) *Tan Mug* (i) *Redgray Mug*

Figure 4.6: The objects used to collect the dataset. The first row are the three source containers. The last two rows are the 6 target containers.

to trial without the object accumulating heat and appearing the same temperature as the liquid in the thermal image. The only exception to this was the lid of the *mug*, which was not thermally insulated. It was submersed in cold water between each trial to prevent heat build-up.

We used two different types of target containers, 3 large containers and 3 small. The 3 large containers were the *bowl*, the *fruit bowl*, and the *pan* shown in Figures 4.6d, 4.6e, and 4.6f. The 3 small containers were the *small bowl*, the *tan mug* and the *redgray mug* shown in 4.6g, 4.6h, and 4.6i. Each target container was swapped out at the end of each trial to allow it time to dissipate the heat from the hot liquid.

4.3.4 Data Collection

We collected 1,009 pouring trials with our robot. For every trial on our robot we collected color, depth, and thermal images. Additionally we collected 20 pouring trials for evaluating our methodology's generalization ability with objects not present in the training set. We collected 648 specifically for the perception task in this chapter while the remaining 361 were collected while performing the pouring experiments in the next chapter[4].

For the 648 pouring trails collected here on our robot we did the following. We fixed the robot's gripper over the target container and placed the source container in the gripper pre-filled with a specific amount of liquid. The robot controlled the angle of the source by rotating its wrist joint. We systematically varied 6 variables:

- *Arm* - left or right
- *Source Container* - *cup*, *bottle*, or *mug*
- *Target Container* - *bowl*, *fruit bowl*, or *pan*

[4]The experiments in the following chapter were conducted contemporaneously with the experiments in this chapter and so we were able to add the data collected in those experiments to train the model-free volume estimation networks to our dataset here since it is largely the same format.

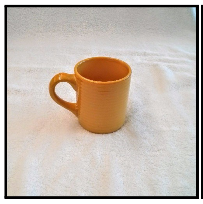
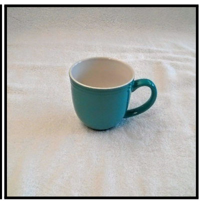
- *Fill Amount* - empty, 30%, 60%, or 90%
- *Trajectory* - partial, hold, or dump
- *Motion* - minimal, moderate, or high

We used both arms, as well as varied the source containers. We also used the 3 large target containers to contrast with the 3 small ones used in the other data (described in the next chapter). In addition to various fill percents, we also included trials with no liquid to provide negative examples (which we use for both training and evaluating our networks). The robot followed three fixed pouring trajectories: one in which it tilted the source to parallel with the ground and then returned to vertical; one in which it tilted the source to parallel with the ground and held it there; and one in which the robot quickly rotated the source to pointing almost vertically down into the target. Finally, we added motion to the data. For minimal motion, the only motion in the scene was the robot's with minimal background motion. For moderate motion, a person moved around in the background of the scene while the robot was pouring. For high motion, a person grasped and held the target container and actively moved it around while the robot poured into it.

Test Data

We also collected 20 pouring trials on our robot to evaluate our methodology's generalization ability. We used the target containers in Figure 4.7 which were not included in the training datasets described in the previous sections. For each object, we recorded 3 trials using the *mug* as the source container, the robot's right arm, and we filled the source initially 90% full. We collected one trial for each of the pouring trajectories described previously (partial, hold, and dump) with minimal background motion. We collected 2 more trials with each object where we fixed the pouring trajectory (fixed as dump) and varied the motion between moderate and high. Overall there were 5 trials per test object for a total of 20 test trials.

(a) *Blue Bowl* (b) *Tan Bowl*

(c) *Gold Mug* (d) *Teal Mug*

Figure 4.7: The target containers used to create the testing set.

4.3.5 Generating the Ground Truth from Thermal Images

We process the thermal images into ground truth pixel labels as follows. First, we normalize the temperature values for each frame in the range 0 to 1. For all frames before liquid appears, we use the normalization parameters from the first frame with liquid. We then threshold each frame at 0.6, that is, all pixels with values lower than 0.6 are labeled *not-liquid* and all pixels higher are labeled *liquid*.

While this results in a decent segmentation of the liquid, we can further improve it by removing erroneously labeled liquid pixels. For example, during some sequences the robot briefly missed the target container, causing water to fall onto the table and be absorbed by the towel. While this is still technically liquid, we do not wish to label it as such because after being absorbed by the towel, it's appearance qualitatively changes. We use the PointCloud Library's plane fitting and point clustering functions to localize the object on the table from the depth image, and we remove points belonging to the table[5] Additionally, for some trials,

[5]We keep points above the lip of the target container in the image so as to not remove the stream of

the lid on the *mug* did not properly cool down between trials, and so for those trials we use a simple depth filter to remove pixels too close to the camera (the source container is slightly closer to the camera than the target).

4.4 Learning Methodology

We utilize deep neural networks to learn the task of *detection*. Specifically, we use fully-convolutional networks (FCNs) [87], that is, networks comprised of only convolutional layers (in addition to pooling and non-linear layers) and no fully-connected layers. FCNs are well suited to the tasks in this paper because they produce pixel-wise labels and because they allow for variable sized inputs and outputs. The following sections describe the different types of inputs and outputs for our networks, as well as the different network layouts.

4.4.1 Network Input

We implemented 5 different types of input images to feed into our networks. The first was the standard RGB image shown Figure 4.8a. This is the type of FCN input most commonly seen in the literature [49, 125], and we use it as the primary type of input for tasks on the simulated dataset. However, since the robot dataset is one tenth the size of the simulated dataset, and thus is more prone to overfitting, we also desired to evaluate other types of images that may help counteract this tendency to overfit. The most obvious type of image is grayscale, which was very commonly used in computer vision methods prior to deep networks [31]. Figure 4.8b shows a grayscale version of the RGB image in Figure 4.8a.

Inspired by prior work [163], we also evaluated optical flow as an input to the networks. We computed the dense optical flow for a given frame by calling OpenCV's `calcOpticalFlowFarneback` on that frame and the frame immediately prior (for the first frame we used the following frame instead). For the parameters to the function `calcOpticalFlowFarneback`, we set the number of pyramid levels to 3 and the pyramid scale to 0.5, the window size to 15 and the

liquid as it transfers form the source to the target.

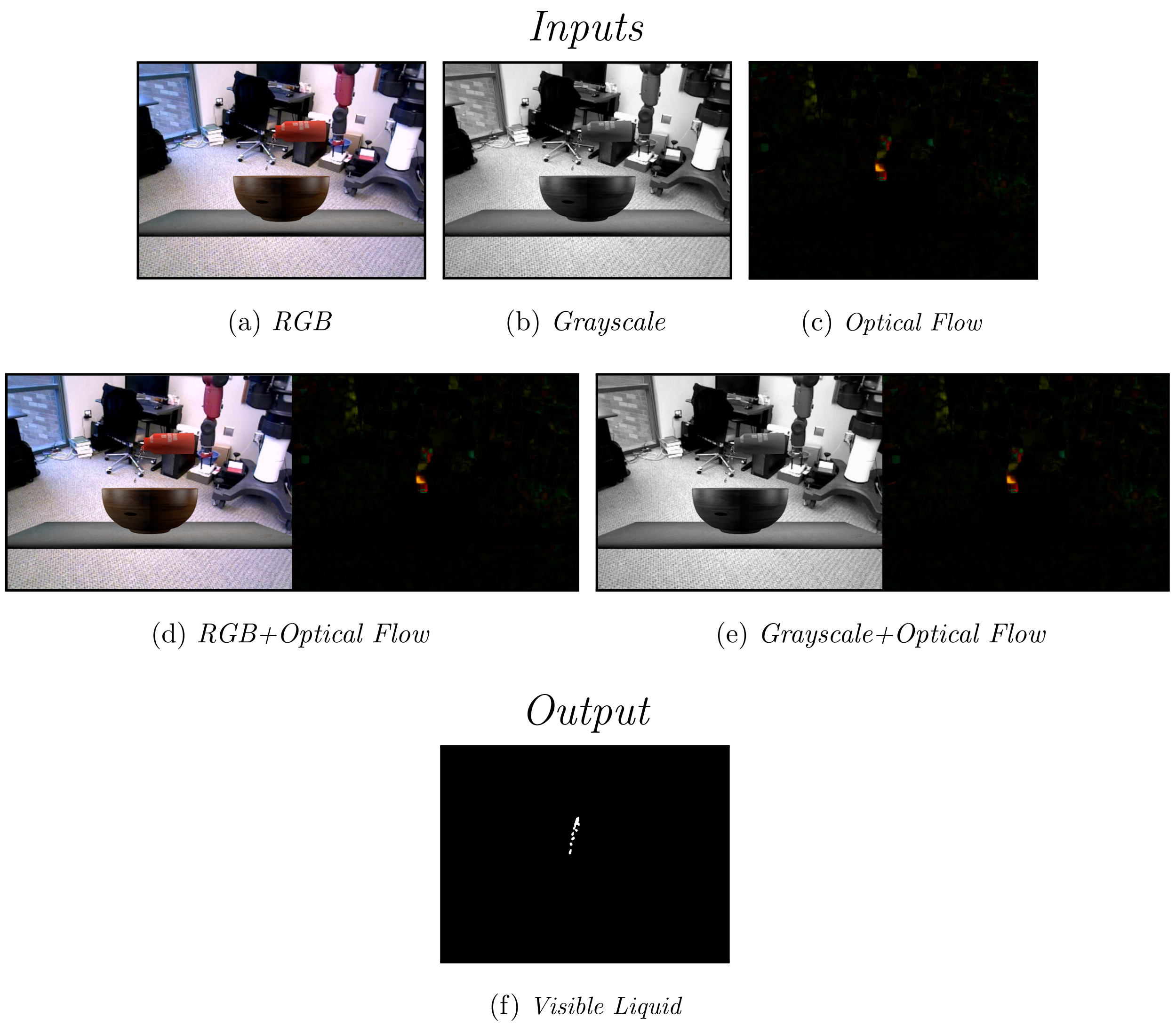


Figure 4.8: Different images of the same frame from the same sequence. The upper part of this figure shows the different types of network inputs (RGB, grayscale, optical flow, RGB+optical flow, and grayscale+optical flow). The lower part shows the desired network outputs (pixel labels for *liquid* and *not-liquid*).

number of iterations to 3, and the pixel neighborhood size to 5 with a standard deviation of 1.2. Besides calling `calcOpticalFlowFarneback`, we did not perform any other filtering or smoothing on the optical flow output.

The output of the dense optical flow was an xy vector for each pixel, where the vector was the movement of that feature from the first frame to the second. We converted each vector to polar coordinates (angle and magnitude), and further converted the angle to the sine and cosine values for the angle, resulting in three values for each pixel. We store the resulting vectors in a three channel image, where the first channel is the sine of each pixel's angle, the second is the cosine, and the third is the magnitude. An example is shown in Figure 4.8c (converted to HSV for visualization purposes, where the angle is the hue and the magnitude is the value). While [163] showed that optical flow at least correlates with moving liquid, it is not clear that flow by itself provides enough context to solve the *detection* problem. Thus we also evaluate combining it with RGB (Figure 4.8d) and grayscale (Figure 4.8e).

4.4.2 Network Output

The desired output of the network for *detection* is the locations of the visible liquid in the scene. An example of this is shown in Figure 4.8f. Note that in the case of Figure 4.8f, most of the liquid is occluded by the containers, so here the robot is detecting primarily the flow of liquid as it transfers from the source to the target container. The output of each network is a pixel-wise label confidence image, i.e., for each pixel, the network outputs its confidence in $[0, 1]$ that that pixel is either *liquid* or *not-liquid*.

4.4.3 Network Layouts

All of the networks we use in this paper are fully-convolutional networks (FCNs). That is, they do not have any fully-connected layers, which means each intermediate piece of data in the network maintains the image structure from the original input. This makes FCNs well-suited for tasks which require pixel labels of the pixels from the input image, which our task of *detection* requires. Additionally, they allow variable sized input and outputs, which we take advantage of during training of our networks (described later in the evaluation section).

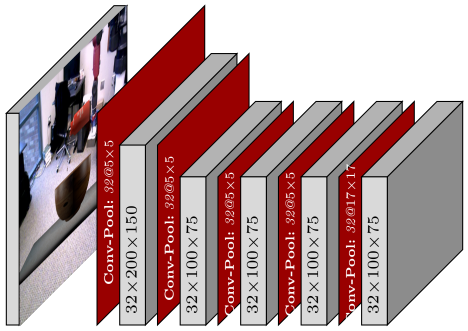
We use the Caffe deep learning framework [58] to implement our networks

Input Blocks

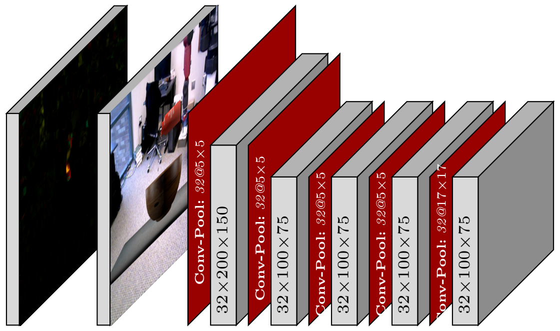
Each network we implement is built from one or more input blocks. Input blocks are combinations of network layers with different types of input. Essentially each is the beginning part of an FCN. We split our description of our neural networks into input blocks and network types (below) to simplify it. We combine our different types of input blocks with our different network types to create a combinatorially larger number of networks, which we then use to solve the task of *detection*.

Figure 4.9 shows the 3 different types of input blocks that we use in this paper. The first (Figure 4.9a) is the standard input block used by most of our networks. It takes as input a single image, which it then passes through 5 conv-pool layers, which apply a convolution, then a rectified linear filter, and finally a max pooling operation. The first two conv-pool layers have a stride of two for the max pooling operation; all other layers have a stride of one. The output of this input block is a tensor with shape $32 \times \frac{H}{4} \times \frac{W}{4}$ where H and W are the height and width of the input image respectively.

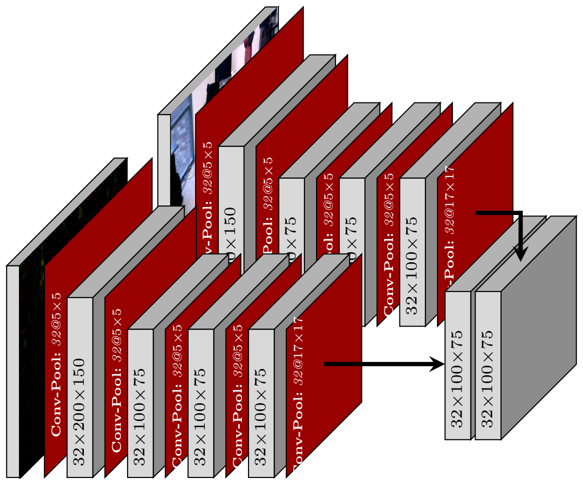
The second two input blocks are used for networks that take two different types of images as input (e.g., RGB and optical flow). Figure 4.9b shows the early-fusion approach, which combines the two images channel-wise and feeds them into a block otherwise identical to the standard input block. Figure 4.9c shows the late-fusion approach, which feeds each image into separate copies of the standard input block, and then concatenates the resulting tensors channel-wise, resulting in a $64 \times \frac{H}{4} \times \frac{W}{4}$ tensor. Some work in the literature has suggested that the late-fusion approach tends to perform better than the early-fusion approach[153], however in this paper we evaluate this premise on our own tasks.



(a) Standard Input Block



(b) Early Fusion Input Block



(c) Late Fusion Input Block

Figure 4.9: The 3 different types of input blocks. The first is used when the network takes only a single type of input; the second two are used when combining two different types of input. Here gray boxes are the feature representations at each level of the network, and the colored squares are the layers that operate on each representation. Gray boxes immediately adjacent indicate channel-wise concatenation.

Network Types

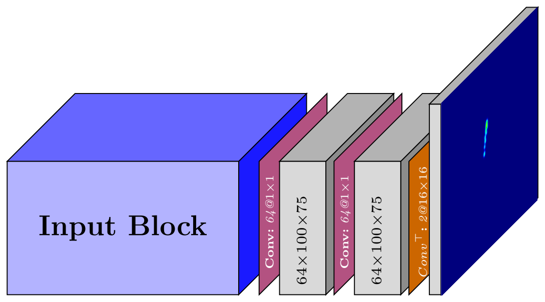
We use 3 different types of networks in this paper:

FCN The first is a standard FCN shown in Figure 4.10a. It takes the output of the input block and passes it through 2 1×1 convolutional layers and a final transposed convolution[6] layer (written as $Conv^\top$ in the figure). The 1×1 convolutional layers take the place of fully-connected layers in a standard neural network. They take only the channels for a single "pixel" of the input tensor, acting similar to a fully-connected layer on a network that takes the image patch of the response region for that pixel. Each 1×1 convolutional layer is followed by a rectified linear filter.

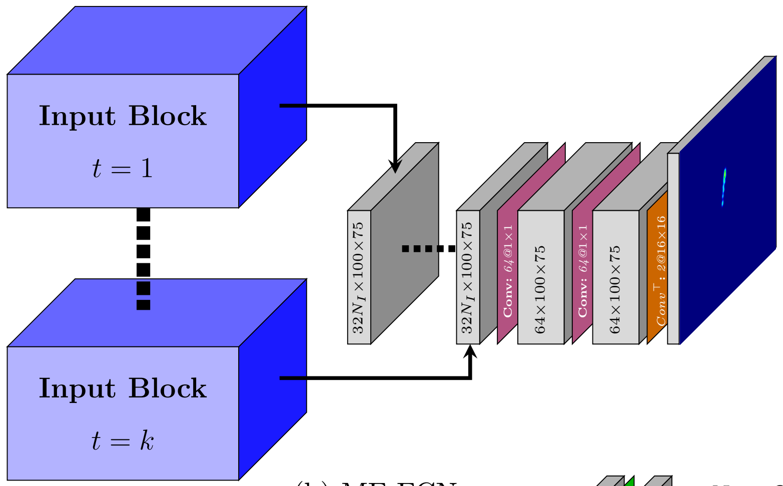
MF-FCN The second network type is a multi-frame FCN shown in Figure 4.10b. It takes as input a series of sequential frames, and so has an input block for each frame. Each input block shares parameters, e.g., the first convolutional layer in the first input block has the exact same kernels as the first convolutional layer in the second input block, and so on. The output tensors of all the input blocks are concatenated together channel-wise. This is then feed to a network structured identical to the structure for the previous network (two 1×1 convolutional layers followed by a transposed convolution layer).

LSTM-FCN The last network type is a recurrent network that utilizes a long short-term memory (LSTM) layer [54] shown in Figure 4.10c. It takes the recurrent state, the cell state, and the output image from the previous timestep in addition to the frame from the current timestep as input. The output tensor from the input block is concatenated channel-wise with the recurrent state, and with the output image from the previous timestep after it has been passed through 3 conv-pool layers. The resulting tensor is then fed into the LSTM layer along with the cell state. The LSTM layer uses the cell state to "gate" the other inputs, that is, the cell state controls how the information in the other inputs passes through the LSTM. The resulting output we refer to as the

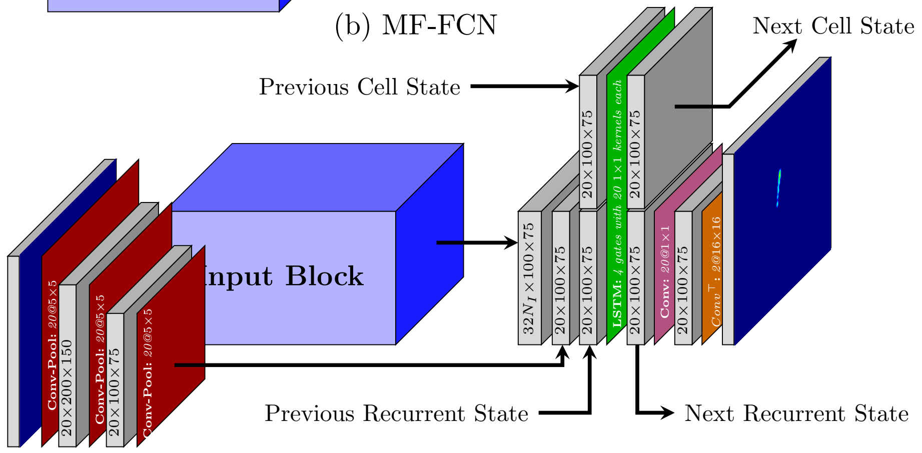
[6]Sometimes referred to in the literature as upsampling or deconvolution.



(a) FCN



(b) MF-FCN



(c) LSTM-FCN

Figure 4.10: The three types of networks we tested. The first is a standard FCN. The second is an FCN that takes in a series of consecutive frames. The final is a recurrent network that uses an LSTM layer to enable the recurrence. As in Figure 4.9, the gray boxes are the feature representations at each level of the network, and the colored squares are the layers that operate on each representation. Gray boxes immediately adjacent indicate channel-wise concatenation (the dashed line in the MF-FCN indicates a concatenation over the range of inputs). N_I indicates the size of the output of the input block, with $N_I = 2$ for the late-fusion blocks and $N_I = 1$ for all other blocks. The LSTM-FCN takes its own output from the previous timestep as input (lower-left), convolves it through 3 layers, and concatenates it with the output of the input block. The LSTM layer is implemented using the layout described in Figure 1 of [41]

"recurrent state" because it is feed back into the LSTM on the next timestep. The LSTM layer also updates the cell state for use on the next timestep. In addition to being used in the next timestep, this recurrent state is also fed through a 1×1 convolutional layer and then a transposed convolution layer to generate the output image for this timestep. To maintain the fully-convolutional nature of our network, we replace all the gates in the LSTM layer with 1×1 convolutional layers. Please refer to Figure 1 of [41] for a more detailed description of the LSTM layer.

4.5 Evaluation

4.5.1 Simulated Data Set

We evaluate all three of our network types on the simulated dataset. For this dataset, we use only single input image types, so all networks are implemented with the standard input block (Figure 4.9a). We report the results as precision and recall curves, that is, for every value between 0 and 1, we threshold the confidence of the network's labels and compute the corresponding precision and recall based on the pixel-wise accuracy. We also report the area-under-curve score for the precision and recall curves. Additionally, we report precision and recall curves for various amounts of "slack," i.e., we count a positive classification as correct if it is within n pixels of a true positive pixel, where n is the slack value. This slack evaluation allows us to differentiate networks that are able to detect the liquid, albeit somewhat imprecisely, versus networks that fire on parts of the image not close to liquid.

We evaluate our networks on two subsets of the simulated dataset: the *fixed-view* set and the *multi-view* set. The *fixed-view* set contains all the data for which the camera was directly across from the table (camera azimuth of 0 or 180 degrees) and the camera was level with the table (low camera height), or 1,266 of the pouring sequences. Due to the cylindrical shape of all the source and target containers, this is the set of data for which the mapping from the full 3D state of the simulator to a 2D representation is straightforward, which is useful for our networks as they operate only on 2D images. The *multi-view* set contains all data from the simulated dataset, including all camera viewpoints. The mapping from 3D to 2D for this set is not as straightforward.

We trained all three networks in a similar manner. Due to the fact that the vast majority of pixels in any sequence are *not-liquid* pixels, we found that trying to train directly on the full pouring sequences resulted in networks that settled in a local minima classifying all pixels as *not-liquid*. Instead, we first pre-train each network for 61,000 iterations on crops of the images and sequences around areas with large amounts of liquid (due to the increased complexity of the LSTM-FCN, we initialize the pre-training LSTM-FCN with the weights of the pre-trained single-frame FCN). We then train the networks for an additional 61,000 iterations on full images and sequences. This is only possible because our networks are fully-convolutional, which allows them to have variable sized inputs and outputs. Additionally, we also employ gradient weighting to counteract the large imbalance between positive and negative pixels. We multiply the gradient from each *not-liquid* pixel by 0.1 so that the error from the *liquid* pixels has a larger effect on the learned weights.

The full input images to our networks were scaled to 400×300. The crops taken from these images were 160×160. The single-frame networks were trained with a batch size of 32. The multi-frame networks were given a window of 32 frames as input and were trained with a batch size of 1. The LSTM networks were unrolled for 32 frames during training (i.e., the gradients were propagated back 32 timesteps) and were trained with a batch size of 5. We used the mini-batch gradient descent method Adam [62] with a learning rate of 0.0001 and default momentum values. All error signals were computed using the softmax with loss layer built-into Caffe [58].

4.5.2 Robot Data Set

For the robot dataset, we evaluate our networks on the task of *detection*. However, *detection* on the robot dataset is more challenging than on the simulated dataset as there is less data to train on. This is a general problem in robotics with deep learning. Deep neural networks require vast amounts of data to train on, but it is difficult to collect this much data on a robot. While there have been some proposed solutions for specific problems [82, 152], there is no generally accepted methodology for solving this issue. Here we evaluate utilizing different types of input images to help prevent the networks from overfitting on the smaller amount of data.

Specifically, we train networks for each of the following input types (with the corresponding input block in parentheses):

- RGB (standard input block)
- Grayscale (standard input block)
- Optical Flow (standard input block)
- RGB+Optical Flow (early-fusion input block)
- Grayscale+Optical Flow (early-fusion input block)
- RGB+Optical Flow (late-fusion input block)
- Grayscale+Optical Flow (late-fusion input block)

We train LSTM networks on all of these different types of inputs, as well as the single-frame networks since they are necessary to initialize the weights of the LSTM networks. We use the same learning parameters and training methodology as on simulated data (pre-training on crops, gradient weighting, etc.). For brevity, we report our results as area under the curve for the precision and recall curves for each network.

Unlike for the simulated dataset, where the train and test sets are created by dividing the dataset, for the robot dataset, we created an explicit test set. To test the robot's generalization ability, we used target containers that did not appear in the train set. We train all networks on the entire dataset and test on this explicit test set. To gauge the extent to which our networks overfit to their training set, we report the performance of the networks on both the train set and the test set.

4.5.3 Baseline for the Robot Data Set

For comparison, we implement as a baseline the liquid detection methodology described in [163] for the *detection* task on the real robot dataset. We briefly describe that implementation here. For each image in a sequence, we compute the dense optical flow using the same methodology as for the neural network method. Next, we compute the magnitude of the flow vector for each pixel, and create a resulting flow magnitude image. We then perform the following steps to filter the image as described in Section II.A of [163]:

1. Erode the image with a square kernel of size 3.
2. Dilate the image with a square kernel of size 3.
3. Apply a temporal filter with size 5 to each pixel, replacing the value in the pixel with the OR of all the pixels covered by the filter.
4. Dilate the image with a square kernel of size 7.
5. Erode the image with a square kernel of size 11.
6. Dilate the image with a square kernel of size 13.
7. Convolve the image with a 12×1 filter (12 pixels high, 1 wide) where each value in the filter is 1/12.
8. Use the result of the prior step to apply as a mask to the result of step 3.
9. Apply the same filter to the result as in step 7.
10. Scale the magnitudes in the resulting image to be in the range 0 to 1.

Note that some of the hyper parameters we used are adjusted from the values used in [163] to account for the difference in image sizes (640×480 vs. 400×300 in this paper).

There are two primary differences between our implementation and the implementation in [163]. The first is the way in which background motion is removed. In that paper, the authors utilized stereo RGB cameras to localize the optical flow in 3D, and then fixed a region of interest around the liquid, removing all motion not in that region. In our work, we use a single camera, however our camera also uses structured infrared light combined with an infrared camera to determine the depth of each point in the image. In order to remove background motion, we generate a mask by including only pixels whose value is closer than one meter from the camera. We then smooth this mask by eroding, then dilating twice, then eroding again, all with a square kernel of size 7. This mask is applied to the optical flow before applying the filter steps above.

The second difference between our implementation and that in [163] is that, in order to be comparable to our methodology, it must compute a distribution over the class labels, rather than a single label. In [163] they compute only a binary mask for each image. However, in the following section we utilize precision-recall curves to compare our methods, which requires a probability distribution over class labels to compute. We approximate this distribution using the magnitude of the flow at each pixel, that is, the more a pixel is moving, the more likely it is liquid.

4.6 Results

4.6.1 Simulated Data Set

Figures 4.11 and 4.12 show the results of training our networks for the *detection* task on the simulated dataset. Figure 4.11 shows the output of each network on example frames. From this figure it is clear that all networks have the ability to at least detect the presence of liquid. However, it is also clear that the MF-FCN is superior to the single-frame FCN, and the LSTM-FCN is superior to the MF-FCN. This aligns with our expectations: As we integrate more temporal information (the FCN sees no temporal information, the MF-FCN sees a small window, and the LSTM-FCN has a full recurrent state), the networks perform better.

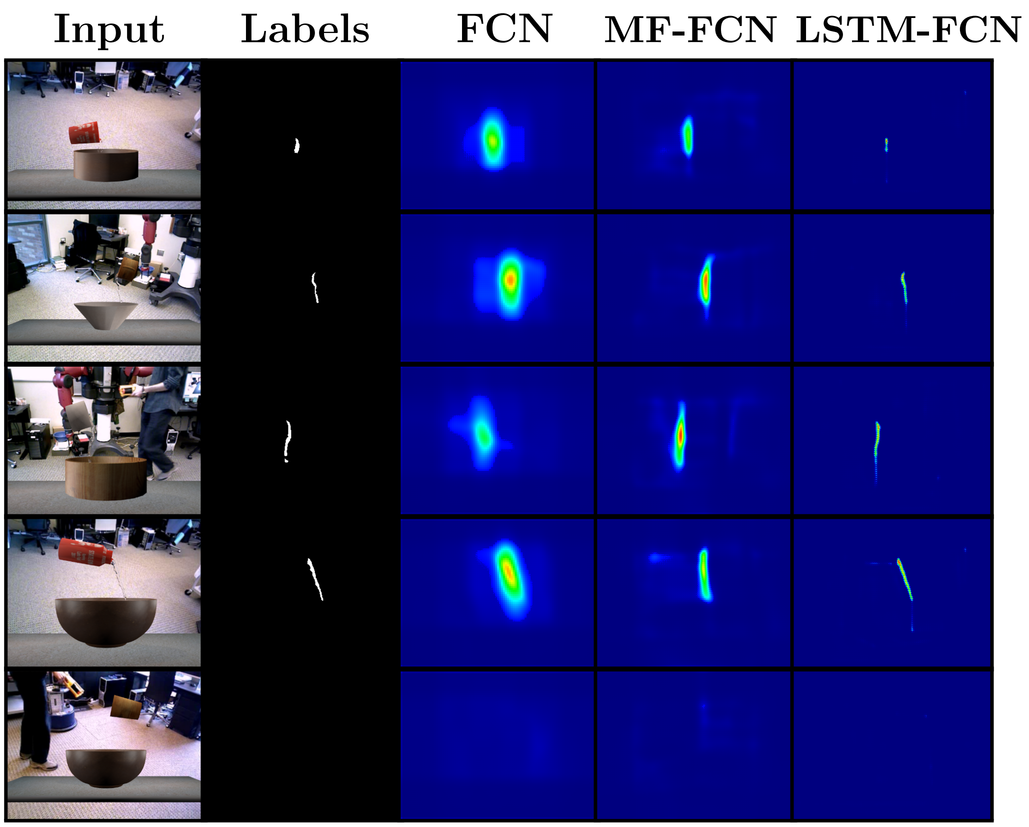
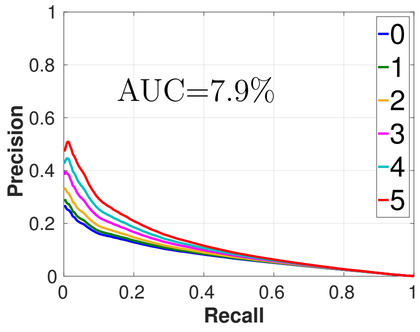
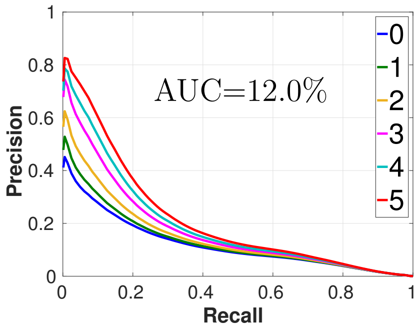


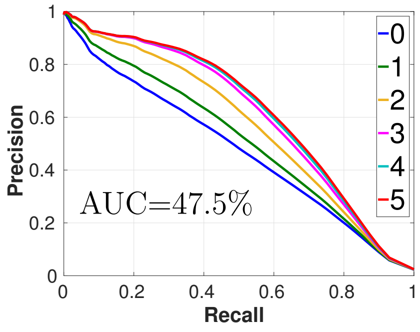
Figure 4.11: Example frames from the 3 network types on the *detection* task on the simulated dataset. The sequences shown here were randomly selected from the test set and the frame with the largest amount of liquid visible was selected. The last sequence was selected to show how the networks perform when no liquid is present.



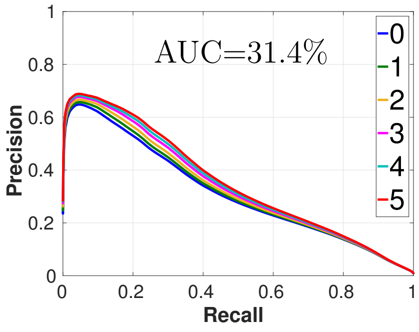
(a) FCN



(b) MF-FCN



(c) LSTM-FCN



(d) MV LSTM-FCN

Figure 4.12: Precision-recall curves for the simulated dataset. The first three show the curves for the three network types on the *fixed-view* subset. The last graph shows the performance of the LSTM network on the *multi-view* subset. The different lines show the different amounts of slack, i.e., how far a positive classification can be from a true positive to still count as correct. The area under the curve (AUC) is shown for the 0 slack curve.

The quantitative results in Figure 4.12 confirm these qualitative results. For reference, all the networks have a very similar number of parameters (414,336, 477,824, and 437,508 for the FCN, MF-FCN, and LSTM-FCN networks respectively), so it is clear that the success of the LSTM-FCN is not simply due to having more parameters and "remembering" the data better, but that it actually integrates the temporal information better.

Since the LSTM-FCN outperformed the other two network types by a significant margin, we evaluated it on the *multi-view* set from the simulated dataset. The performance is shown in Figure 4.12d. Even with the large increase in camera viewpoints, the network is still able to detect liquid with only a relatively small loss in performance. These results combined with the performance of the LSTM-FCN in Figure 4.12c clearly show that it is the best network for performing detection and is the reason we focus on this network for detection on the robot dataset.

4.6.2 Robot Data Set

Figure 4.14 shows example output on the test set of the LSTM-FCN with different types of input. From this figure, it appears that the best performing network is the one that takes as input grayscale images plus optical flow with the early-fusion input block. Indeed, the numbers in the table in Figure 4.13b confirm this. Interestingly, the grayscale + optical flow early-fusion network is the second worst performing network on the train set, but performs the best on the test set. This suggests that the other networks tend to overfit more to the training distribution and as a result don't generalize to new data very well.

The table in Figure 4.13a reflects a similar, albeit slightly different, result for single-frame FCNs. While the grayscale plus optical flow early-fusion network has one of the highest performances on the test set, it is outperformed by the network that takes only optical flow as input. As counter-intuitive as it may seem, this makes some sense. The single-frame FCN does not have the ability to view any temporal information, however since optical flow is computed between two frames, it implicitly encodes temporal information in the input to the network. As we saw in the section on detection on the simulated dataset, temporal

	Optical Flow		
train / **test**	None	Early-Fusion	Late-Fusion
RGB	76.1% / **18.2%**	63.3% / **17.5%**	48.8% / **25.1%**
Gray	70.7% / **22.0%**	57.0% / **30.5%**	73.8% / **24.1%**
Neither	▨	41.3% / **35.1%**	

(a) FCN

	Optical Flow		
train / **test**	None	Early-Fusion	Late-Fusion
RGB	95.1% / **23.7%**	48.3% / **13.4%**	93.1% / **33.1%**
Gray	82.9% / **17.9%**	81.5% / **49.4%**	92.8% / **41.7%**
Neither	▨	82.7% / **37.4%**	

(b) LSTM-FCN

Figure 4.13: The area under the curve (AUC) for the precision-recall curves for the networks for the *detection* task on the robot dataset. The top table shows the AUC for the single-frame FCN; the bottom shows the AUC for the LSTM-FCN. The tables show the AUC for different types of input, with rows for different types of image data (RGB or grayscale) and the columns for different types of optical flow (none, early-fusion, or late-fusion). Each cell shows the AUC on the train set (upper) and the AUC on the test set (lower), all computed with 0 slack.

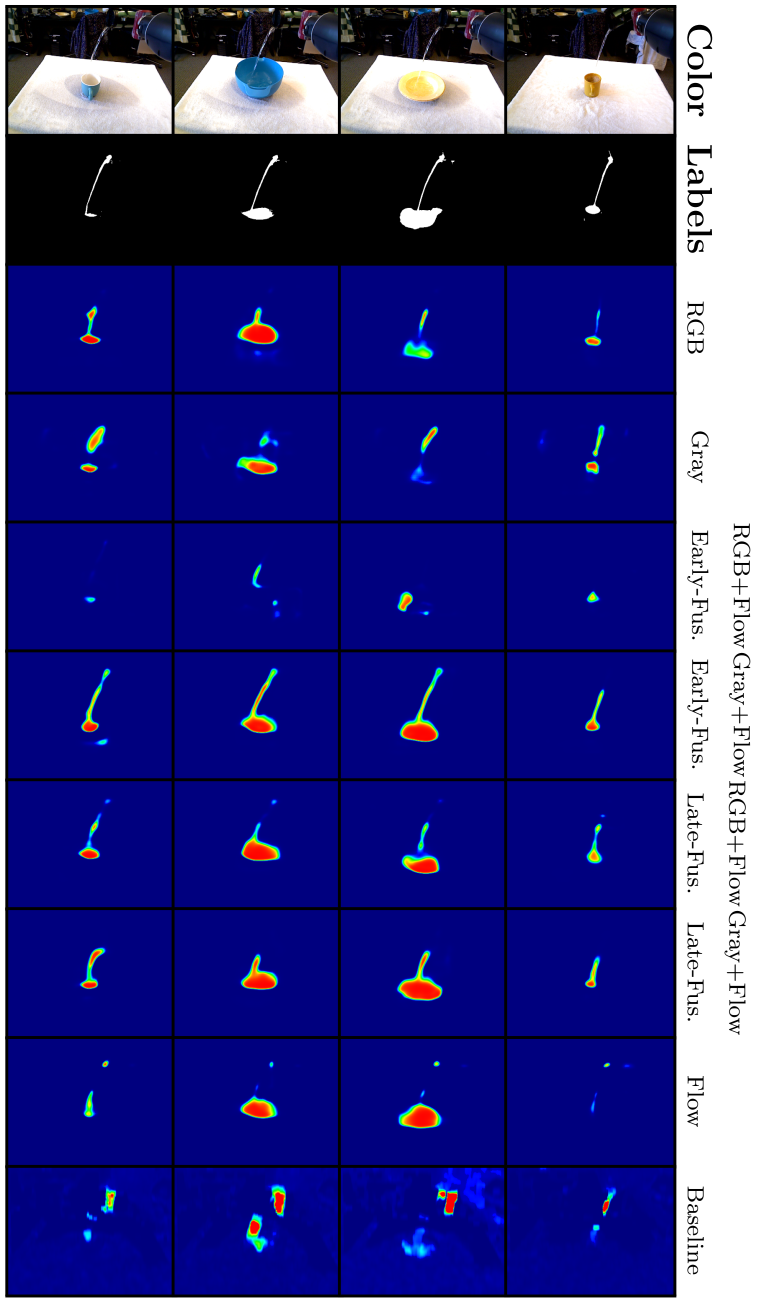


Figure 4.14: Example frames for the LSTM-FCN for the *detection* task on the robot dataset with different types of input images. The first row shows the color image for reference and the row immediately below it shows the ground truth. All these images are from the test set with target object not seen in the train set. The last row shows the output of the baseline.

information is very important for the *detection* task and the network that takes only optical flow is forced to only use temporal information, thus allowing it to generalize to new data better. In the case of the LSTM-FCN, this effect is less pronounced because the network can store temporal information in its recurrent state, although performance of the optical flow only network is still better than performance of the networks that do not use optical flow in any way.

Baseline Comparison

We also computed the performance of the baseline method based on the methodology of [163]. It achieved 5.9% AUC on the training set and 8.3% AUC on the testing set. The last row of Figure 4.14 shows some examples of the output of the baseline. While it is clear from the figure that the baseline is at least somewhat able to detect liquid, it does not perform nearly as well as the neural network based methods. However, it is important to note that this method was developed by [163] for a slightly different task in a slightly different environment and using stereo cameras rather than monocular, so it would not be expected to perform as well on this task. Nonetheless, it still provides a good baseline to compare our methods against.

The biggest advantage of the baseline method over learning-based methods is it's resilience to overfitting due to its lack of trained parameters. However, this lack of learning also means it can't adapt to the problem as well. Inspired by the resilience of the baseline method, we combined it with our deep neural network architectures to soften the effect of overfitting while maintaining the adaptability of learning-based methods. As shown in Figure 4.13b, the methods using optical flow as an input tended to have a smaller disparity between their training set and testing set performance. While this didn't completely alleviate all overfitting, it is clear that combining these two methods is superior to using either alone.

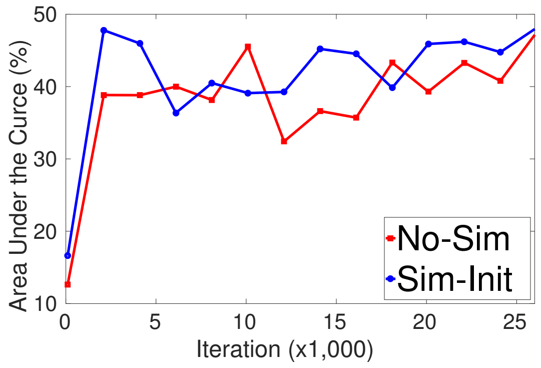


Figure 4.15: The area under the curve on the test set at each iteration of training. The red line shows the performance of the LSTM-FCN trained solely on the robot dataset; the blue line shows the performance of the LSTM-FCN initialized from the simulated dataset.

Initializing on Simulated Data

We evaluated whether or not the simulated data set, with its larger size, could be used to pre-train the weights of a network that would then be trained on the robot data. Since the LSTM-FCN with grayscale plus optical flow early-fusion as input generalized the best in the previous section, we trained another LSTM-FCN on the same type of input. However, instead of pre-training it on cropped images from the robot dataset, we went through the full training process for a detection network on the simulated dataset, and used those weights to initialize this network, which was then trained on the robot dataset. The networks converged to the same performance after 61,000 iterations of training. Figure 4.15 shows the performance of both the network not initialized with any simulated data as compared to the performance of this network on the test set at each iteration. The network initialized with simulated data does seem to converge slightly faster, although not by a large amount.

4.7 Discussion

In this chapter, we showed how a robot can solve the task of detection for liquids using deep learning. We evaluated 3 different network architectures, FCN, MF-FCN, and LSTM-FCN, all of which integrated different amounts of temporal information. We also evaluated eight different types of input images to our networks, including RGB and grayscale combined with optical flow. We tested these networks on both data we generated in a realistic liquid simulator and on data we collected from a real robot.

Our results clearly show that integrating temporal information is crucial for perceiving and reasoning about liquids. The multi-frame FCN was able to outperform the single-frame FCN because it incorporated a window of frames, giving it more temporal information. Furthermore, the LSTM-FCN is able to learn to remember relevant information in its recurrent state, enabling it to outperform the MF-FCN since it keeps information much longer than the fixed window of the MF-FCN.

The results also showed that, for the purposes of generalizing to new objects and settings, standard RGB images lead to overfitting and are not as well suited as images converted to grayscale and early-fused with optical flow. Networks trained on RGB images tended to perform very well on sequences drawn from the same distribution as their training set, but their performance dropped considerably when those sequences were drawn from a slightly different setting. However, while networks trained on grayscale early-fused with optical flow did not reach the same level of performance on data taken from the training distribution, their generalization to new settings was significantly better.

In this chapter we discovered many new insights on perceiving liquids. First, we found that deep learning can be applied to address this challenging perception task with the right architecture. Second, we introduced a novel technique using a thermographic camera and hot water to automatically generate ground truth labels for our real robot dataset. Third, we investigated different deep network structures and showed through experimental evaluation how different types and combinations of inputs affect a networks ability to solve the detection task. In the following chapter we use these insights to apply our perception model to the pouring control task.

4.7.1 Practical Takeaways

While generating the results for this chapter, we also gained several technical insights. The first major issue we encountered in this research was the negative label bias. In the images we used, the vast majority of the pixels were *non-liquid*. When we trained networks from scratch on these images, they all settled in a local minima, labelling all pixels as *non-liquid* and none as *liquid*, even when they contained liquid. We multiplied the loss for *liquid* pixels by a large value to encourage the network to not misclassify them, but by itself this did not have an effect (in this chapter we used a 10:1 ratio for *liquid* pixel error versus *non-liquid* pixel error, but we increased this ratio as high as 10,000:1 with no noticeable effect when used alone). What did work was first training the networks on crops around large amounts of liquid pixels, which balanced the ratio of *liquid* to *non-liquid* pixels. This combined with weighting the loss on *liquid* pixels higher had a significant effect on performance. On the other hand, changing variables such as the number of kernels in a layer, the size of the kernels, or even the number of convolutional layers in network had little effect on the performance of the networks.

Since these results were generated, the technique of domain randomization [149] has become a popular technique for enabling sim-to-real transfer. This is the technique of training a neural network in a simulation environment while varying the visual properties of said environment. This forces the network to operate under a wide variety of visual conditions, which then allows it to work when applied in a real environment. In this chapter we applied domain randomization (before it was called that) to our generation of the simulated dataset. However, we found that it had only a small effect on the performance of our network on the real data (it converged slightly faster when training on real data, but did not directly transfer). While deep networks require a lot of data to train, we generated 4.5 million images or about 2.5TB of data. For the LSTM-FCN, with a batch size of 5 and unroll of 32 timesteps during training for 61,000 iterations, it would have been trained on approximately 9.7 million images, which means it would have seen each image in our dataset about twice. This makes it unlikely that generating more data would have improved the sim-to-real transfer. Ultimately, what we discovered here was that varying the visual properties of the data can only help a neural network so much; if the visual parameter space does not encompass the real environment, then it will be more difficult for sim-to-real transfer.

Chapter 5

LEARNING TO SEE & POUR: COMBINING MANIPULATION AND PERCEPTION

In the previous chapters we examined two different components of a robotic liquid control pipeline. In chapter 3 we looked at how a robot can pour precise amounts of liquid given a means to measure the rate of transfer from the source container to the target. In chapter 4 we examined several methods for using fully convolutional neural networks (FCNs) to solve the *detection* task, i.e., labelling each pixel in an RGB image as *liquid* if it is liquid or *not-liquid* if not. In this chapter we reexamine the pouring task, however here we combine it with the perception pipeline we developed in chapter 4.

Here we introduce a framework to allow robots to pour specific amounts of liquid using only RGB feedback. The intuition behind our approach is based on the insight that people strongly rely on visual cues when pouring liquids. For example, a health study revealed that the amount of wine people pour into a glass is strongly biased by visual factors such as the shape of the glass or the color of the wine [156]. We thus propose a framework that uses visual feedback in a closed-loop pouring controller. Specifically, we train a deep neural network structure to estimate the amount of liquid in a cup from raw visual data. Our network structure has two stages. In the first stage, a network detects which pixels in a camera image contain water. The output of the detection network is fed into another network that estimates the amount of liquid already in the container. This amount is used as real-time feedback in a PID controller that is tasked to pour a desired amount of water into a cup.

To generate labeled data needed for the neural networks, we use the experimental setup

[0]The contents of this chapter were published in [132].

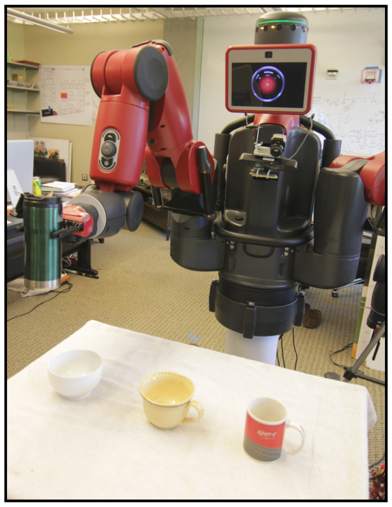


Figure 5.1: The Baxter robot used in our experiments. In its right gripper it holds the cup used as the source container. On the table in front of the robot are the three target containers (from left to right): the small bowl, tan mug, and redgray mug.

from the previous chapter that uses a thermal camera calibrated with an RGBD camera to automatically label which pixels in the color frames contain (heated) water. Experiments with a Baxter robot pouring water into three different containers (two mugs and one bowl) indicate that this approach allows us to train deep networks that provide sufficiently accurate volume estimates for the pouring task.

5.1 Technical Approach

5.1.1 Task Overview

In the experiments in this chapter, the robot is tasked with pouring a specific amount of liquid from a source container into a target container. This task is more difficult than prior work on robotic pouring which primarily focuses on pouring all the contents of the source container into the target container, whereas we focus on pouring only a limited amount from the source containing an unknown initial amount of liquid. To accomplish this, the robot

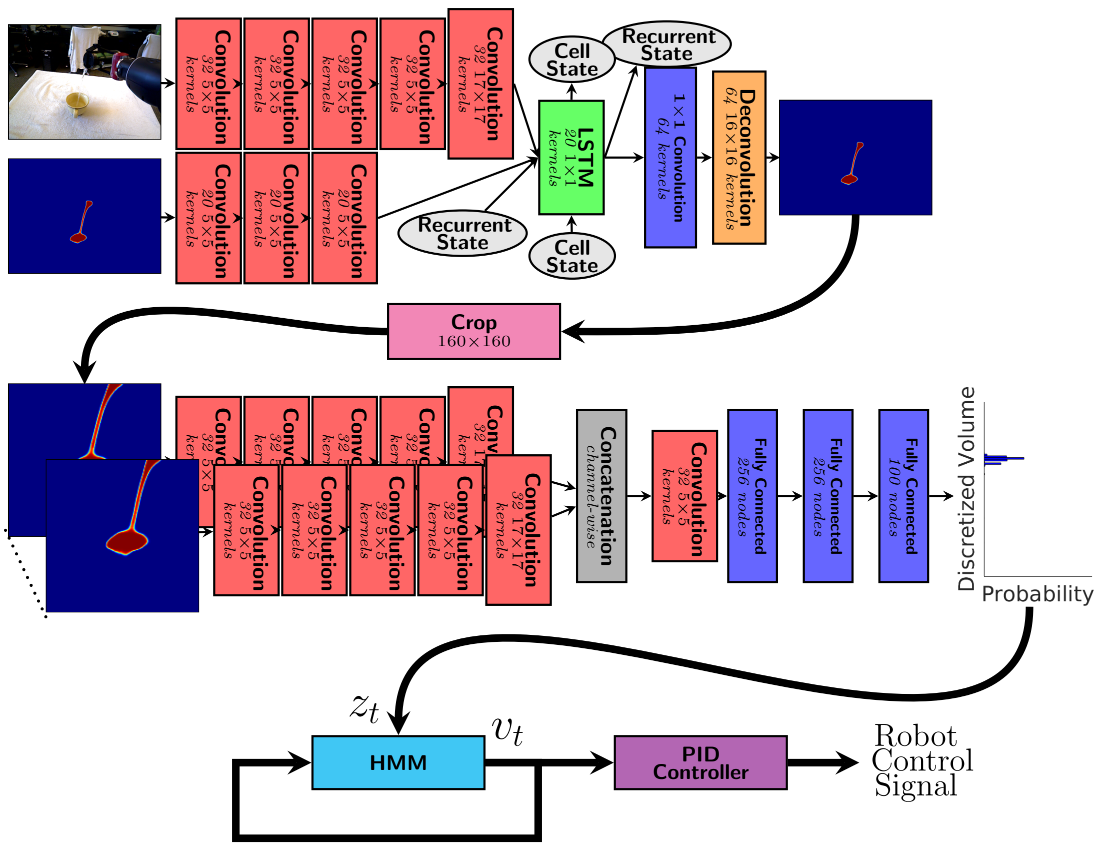

Convolution 32 5×5 kernels
Convolution 32 17×17 kernels
Convolution 20 5×5 kernels
Cell State
Recurrent State
LSTM 20 1×1 kernels
1×1 Convolution 64 kernels
Deconvolution 64 16×16 kernels
Crop 160×160
Concatenation channel-wise
Fully Connected 256 nodes
Fully Connected 100 nodes
Discretized Volume
Probability
z_t
v_t
HMM
PID Controller
Robot Control Signal

Figure 5.2: The entire robot control system using the recurrent neural network for detections and the multi-frame network for volume estimation. The recurrent detection network (top) takes both the color image and its own detections from the previous time step and produces a liquid detection heatmap. The multi-frame network (center) takes a sequence of detections cropped around the target container and outputs a distribution over volumes in the container. The output of this network is fed into a HMM, which estimates the volume of the container. This is passed into a PID controller, which computes the robot's control signal.

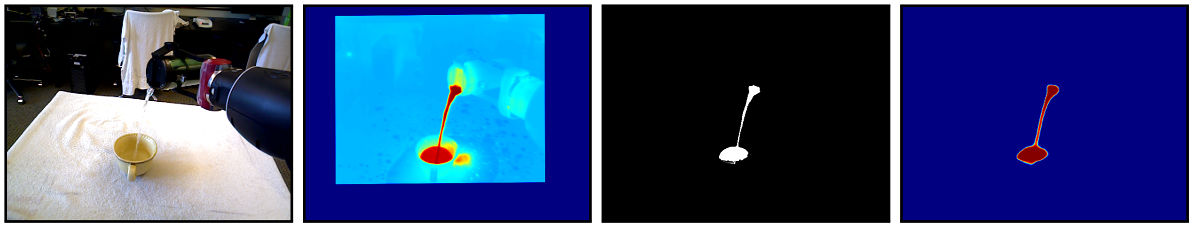


Figure 5.3: An example frame from a pouring sequence. The left image shows the color camera from the robot's perspective, the left-middle image shows a heatmap of the thermal camera after it has been registered to the color camera, the right-middle image shows the water labeled by thresholding the thermal image, and the right image shows the output of the detection network based on the RGB input.

must use visual feedback to continuously estimate the current volume of liquid in the target container. Our approach has 3 main components: First the robot detects which pixels in its visual field are liquid and which are not. Next the robot uses these detections to estimate the volume of liquid in the target container. Finally, the robot feeds these volume estimates into a controller to pour the liquid into the target. Figure 5.2 shows a diagram of this process. We structure the problem in this manner as opposed to simply training one end-to-end network as it allows us to train and evaluate each of the individual components of the system, which can give us better insight into its operation.

5.1.2 Pixel-Wise Liquid Detection

In order for the robot to solve this task, the robot must first classify each pixel in the image as *liquid* or *not-liquid*. In the prior chapter we developed two methods for acquiring these pixel labels: a thermographic camera in conjunction with heated water, and a fully-convolutional neural network[87]. While the thermal camera works well for generating pixel labels, it is also rather expensive and must be registered to an RGBD sensor. We also evaluate using a color camera to acquire the pixel labels. To do this, we use the deep neural network from the previous chapter. Specifically, we use the LSTM-FCN (i.e., the recurrent network) because it had the best performance. Additionally, we use RGB images as input. The results in chapter 4 showed that the input type that had the best generalization performance was the grayscale with optical flow early-fusion. However the RGB input had the highest performance when evaluated on objects it had seen before. Here we assume the robot has a chance to interact with the target containers prior to evaluation, so we use RGB as input. The network layout is shown in the upper part of figure 5.2.

5.1.3 Volume Estimation During Pouring Sequences

We propose two different methods for estimating the volume of liquid in a target container. The first is a model-based method, which assumes we have access to a 3D model of the target container and infers the height of the liquid based on the camera pose and binary pixel labels. The second is a model-free method that trains a neural network to regress to the volume of liquid in the target container given labeled pixels.

Filtering using a HMM

Before describing our volume estimation methods, we first describe our filtering method, which will make our notation in the following sections clearer. Because of the temporal nature of the task, we utilize a hidden Markov model (HMM) to filter the volume estimates over time. Let t be the current timestep, v_t be the volume of liquid in the target container at time t (the hidden state in the HMM), and let z_t be the observation at time t (described in detail in the following sections). To compute the probability distribution over v_t we can apply Bayes rule. HMMs make the Markovian assumption, that is, v_t is conditionally independent of all prior observations and states given v_{t-1} and z_t is conditionally independent of all prior observations and states given v_t. Thus we can write the posterior as

$$P(v_t|z_t, v_{t-1}) \propto P(z_t|v_t)P(v_t|v_{t-1}).$$

For this chapter, we represent the distribution over v as a histogram over a fixed range, and so the transition probability $P(v_t|v_{t-1})$ is a summation over the bins in the histogram

$$P(v_t|v_{t-1}) = \sum_i P(v_t|v_{t-1} = i)P(v_{t-1} = i). \tag{5.1}$$

The transition probability $P(v_t|v_{t-1})$ is inferred from the training data.

The following two sections describe how we compute the observation probability $p(z_t|v_t)$, which varies for the model-based and model-free methods. During task execution, we compute the volume of liquid at a given timestep t by taking the median over the posterior distribution on v_t[1].

Model-Based Volume Estimation

Our model-based method for estimating the volume of liquid in a target container assumes we have a 3D model of the container and that we can use the pointcloud from our RGBD sensor to find its pose in the scene. The observation z_t for this method is the set of pixel-wise liquid labels for the image at time t, computed as described in section 5.1.2. Intuitively, to compute the observation probabilities, this method compares the actual observation z_t to what the robot would expect to see if the volume of liquid in the container were v_t.

More formally, we compute $P(z_t|v_t = i)$ as follows. First, we assume conditional independence between every pixel $p_t^j \in z_t$. Thus the observation probability becomes

$$P(z_t|v_t = i) = \prod_j P(p_t^j|v_t = i).$$

For this product, we only consider the set of pixels that view the inside of the container, i.e., the set of pixels that could potentially be labelled liquid. An example of this is shown in Figure 5.4. The dashed lines show the pixels whose rays intersect the interior of the container, whereas the gray areas represent the pixels whose rays do not. Since the pixels

[1]We also evaluated using other methods such as the expectation or maximum likelihood, but we empirically determined that median produced more stable and less skewed estimates, although all the methods only had minor differences.

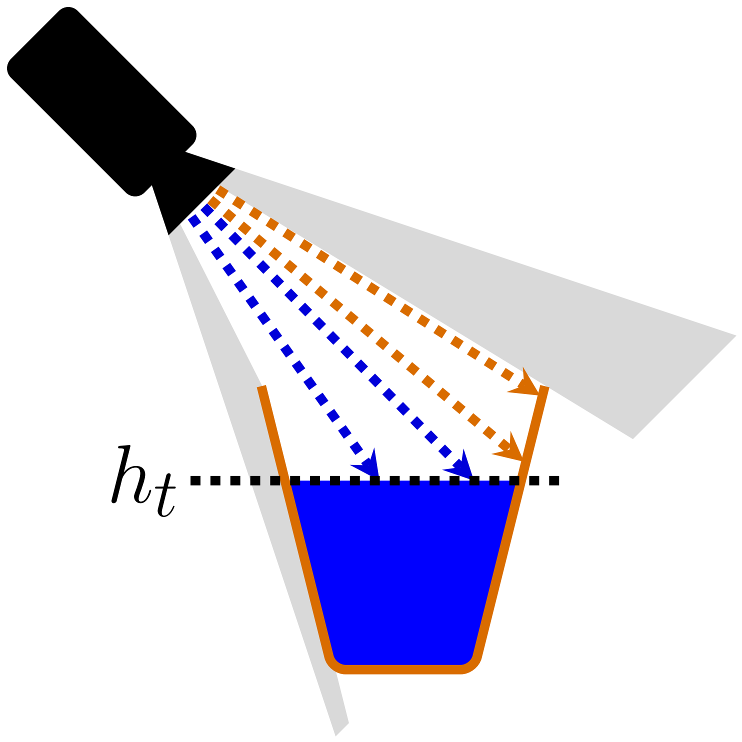


Figure 5.4: Diagram of the camera looking into the target container. The blue shows where the liquid is expected to be, given a volume and the corresponding fill height, h_t. The blue lines show pixels expected to be classified as *liquid* and the orange lines show pixels that are expected to be classified as *not-liquid*. The gray shows pixels outside the container and not used by our algorithm.

in the gray area can not see liquid in the container, they have no effect on the observation probability and thus are not considered.

To compute $P(p_t^j | v_t = i)$, we use the 3D mesh of the container and fill it with $v_t = i$ volume of liquid. We then project that liquid back into the camera to get the expected pixel label $\widehat{p}_t^j$. The observation probability is then

$$\begin{aligned} P(p_t^j | v_t = i) = & P(p_t^j | \widehat{p}_t^j = \textit{liquid}) P(\widehat{p}_t^j = \textit{liquid} | v_t = i) + \\ & P(p_t^j | \widehat{p}_t^j = \textit{not-liquid}) P(\widehat{p}_t^j = \textit{not-liquid} | v_t = i). \end{aligned}$$

To compute $P(\widehat{p}_t^j | v_t = i)$, we assume that the liquid is resting level in the container[2]. At rest, the surface of the liquid will be parallel to the ground, and so we find the height h_t of the surface. We place a plane parallel to the ground at h_t and check whether the ray from pixel p_t^j intersects that plane prior to intersecting the 3D mesh of the container. We set $P(\widehat{p}_t^j = \textit{liquid} | v_t = i)$ to be 1 if the ray intersects the plane first (and 0 otherwise), and we set $P(\widehat{p}_t^j = \textit{not-liquid} | v_t = i)$ to be 1 if the ray intersects the mesh first (and 0 otherwise). Figure 5.4 shows an example. The blue dashed lines show pixels that intersect the plane, whereas the orange dashed lines show pixels that intersect the mesh before intersecting the plane. To compute $P(p_t^j | \widehat{p}_t^j)$, we use the following table:

		p_t^j	
		Liquid	*Not-liquid*
$\widehat{p}_t^j$	*Liquid*	90%	10%
	Not-liquid	20%	80%

To compute the height of the surface of the liquid h_t, we use binary search in combination with the signed tetrahedron volume method [167]. That is, given a height h_t, we can compute the volume of the interior of the container below that height using its 3D mesh by applying the method described in [167]. This volume under the plane corresponds to the volume of

[2]While not strictly true throughout the entire duration of a pour, this assumption still allows for a good measurement.

liquid resting in the container. We then perform binary search over the height to find the h_t that corresponds to $v_t = i$. Note that because the distribution over v_t is a histogram with a fixed set of bins, we can precompute the height for each value of v_t for each 3D mesh.

Model-Free Volume Estimation

Our model-free method replaces the object pose inference of the model-based method with a neural network. The neural network takes in pixel labels and produces a volume estimate. We use only the output of the detection network described in section 5.1.2 for the pixel labels, so we directly feed the heatmap over the pixels into the volume estimation network. We also evaluate adding as inputs either the color or depth images, which we append channel-wise to the pixel labels before feeding into the network. We crop the input to the network around the target container.

Formally, the network computes the function $f(z_t) = \tilde{v}_t$, where z_t is the observation and $\tilde{v}_t$ is an estimate of the volume in the container. The estimate $\tilde{v}_t$ is used as the observation in the HMM, i.e., we use $P(\tilde{v}_t|v_t)$ in place of $P(z_t|v_t)$ since $\tilde{v}_t$ is a function of the observation z_t. The output of the network is a distribution over $\tilde{v}_t$. To compute $P(\tilde{v}_t|v_t)$, we sum over all values for $\tilde{v}_t$

$$P(\tilde{v}_t|v_t) = \sum_k P(v_t|\tilde{v}_t = k)P(\tilde{v}_t = k)$$

where $P(\tilde{v}_t = k)$ is computed by the network. The conditional probability $P(v_t|\tilde{v}_t = k)$ is inferred from the training data.

We evaluated three different network architectures: a single-frame convolutional neural network (CNN)[3], a multi-frame CNN, and a recurrent LSTM CNN. We use the Caffe deep learning framework [58] to implement our networks

Single-Frame CNN: The single-frame network is a standard CNN that takes as input a single image. It then passes the image through 5 convolution layers, each of which is followed

[3]Note that for the volume estimation networks, we use CNNs and not FCNs as we do for the *detection* network. This is because the output of the volume estimator is a single value, not a value for each pixel, so a CNN is more suited to the task.

by a max pooling and rectified linear layer. Every layer has a stride of 1 except for the first 3 max pooling layers, which have a stride of 2. It passes the result through 3 fully connected layers, each followed by a rectified linear layer. These last 3 layers are also followed by dropout layers during training, with a drop rate of 10%. The single-frame network (CNN) is similar to the multi-frame network shown in the center row of Figure 5.2, with the exception that it only takes a single frame and does not have the concatenation layer or the convolution layer immediately following it.

Multi-Frame CNN: The multi-frame network (MF-CNN) is shown in the center row of Figure 5.2. It takes as input a set of temporally sequential images. Each image is passed independently through the first 5 layers of the network, which are identical to the first 5 convolutional layers in the single-frame network. Next, the result of each image is concatenated channel-wise and passed through another convolution layer (which is also followed by max pooling and rectified linear layers). This is then fed into 3 fully connected layers, which are identical to the last 3 layers of the single-frame CNN.

Recurrent LSTM CNN: The LSTM-CNN is identical to the single-frame network, with the exception that we replace the first fully connected layer with the LSTM layer. In addition to the output of the convolution layers, the LSTM layer also takes as input the recurrent state from the previous timestep, as well as the cell state from the previous timestep. Each gate in the LSTM layer is a 256 node fully connected layer. Please refer to Figure 1 of [41] for a detailed layout of the LSTM layer.

5.1.4 Robot Controller

For this chapter, we want to investigate whether, given good real-time feedback, pouring can be performed with a simple controller. We place a table in front of the robot, and on the table we place the target container. We fix the source container in the robot's right gripper and pre-fill it with a specific amount of water not known to the robot. We also fix the robot's arm such that the source container is above and slightly to the side of the target container.

To pour, the robot controls the angle of its wrist joint, thus directly controlling the angle of the source container. We use a modified PID controller to execute the pour. This is in contrast to the policy learned via guided policy search we used in chapter 3, which was necessary to handle the unknown, variable sensor delay present there, however that delay is not present here, so a PID controller suffices. For the PID controller, the robot first tilts the container to a pre-specified angle (we use 75 degrees from vertical), then begins running the PID controller, using the difference between the target volume and the current volume in the target container as its error signal, which it uses to set the angular velocity of its wrist joint. Since pouring is a non-reversible task (liquid cannot return to the source once it has left), the integral term does nothing except push the robot to pour faster, so we set its gain to 0[4]. We set the proportional and derivative gains to $\frac{0.01\pi}{180}$ and $\frac{0.2\pi}{180}$ respectively, which we empirically determined to perform well for the pouring task. Once the target volume has been reached, the robot stops the PID controller and rotates the source container until it is vertical once again.

5.1.5 Implementation Details

Finding the Container in the Scene

Both our model-based and model-free methods require finding the target container on the table in front of the robot (though only the model-based needs a 3D model). To find the container, we use the robot's RGBD camera to capture a pointcloud of the scene in front of the robot and then utilize functions in the PointCloud Library (PCL) [128] to find the plane of the table and cluster the points on top of it. To acquire the pose for the model-based method, we use iterative closest points to find the 3D pose of the model in the scene. Next we use this pose to label each pixel in the image as either *inner* (inside of the container), *outer* (outside of the container), or *neither.*

[4]We refer to the controller as a PID controller for easier comprehension by the reader, but it is technically a PD controller.

Generating Ground Truth Pixel Labels

We use a thermal camera in combination with water heated to approximately 93°Celsius to get the ground truth pixel labels for the liquid. To register the thermal image to the color image, we use a paper checkerboard pattern attached to a 61×61 centimeter metal aluminum sheet. We then direct a small, bright spotlight at the pattern, causing a heat differential between the white and black squares, which is visible as a checkerboard pattern in the thermal image. We use OpenCV's built-in function for finding corners of a checkerboard to find correspondence points and compute an affine transformation[5]. We use an adaptive threshold based on the average temperature of the pixels associated with the target container (which includes the pixels for the liquid in the container). The result of this is a binary image with each pixel classified as either *liquid* or *not-liquid*. Figure 5.3 shows a color image, its corresponding thermal image transformed to the color pixel space, and a simple temperature threshold of the thermal image. Note that the thermal camera provides quite reliable pixel labels for liquid detection with minimal false positives

Acquiring Ground Truth Volume Estimates for Training and Evaluation

In order to train our networks in the previous section, and to evaluate both our model-based and model-free methods, we need a baseline ground truth volume estimation. To generate this baseline, we utilize the thermal camera in combination with the model-based method described in section 5.1.3. However, since this analysis can be done *a posteriori* and does not need to be real-time, we can use the benefit of hindsight to improve our estimates, i.e., future observations can improve the current state estimate. While we acknowledge that this method does not guarantee perfect volume estimates, the combined accuracy of the thermal camera and after-the-fact processing yield robust estimates suitable for training and evaluation.

To compute this baseline we replace the forward method for HMM inference described

[5]While there has been prior work on performing full registration between thermal and color images [116], because the depth of the pour is fixed, we opted for this simpler approach. We cannot use the depth sensor because water does not appear on the depth image.

in section 5.1.3 with Viterbi decoding [12]. We replace the summation in equation 5.1 in the computation of the prior $P(v_t|v_{t-1})$ with a *max* to compute the probability of each sequence. We use a corresponding *argmax* to compute the previous state from the current state, starting at the last time step and working backwards. At the last time step, we start with the most probable state. Thus using this method we can generate a reliable ground truth estimate of the volume of liquid in the target container over the duration of a pouring sequence to use for training our learning algorithms and evaluating our methodology.

5.2 Experiments & Results

5.2.1 Robotic Platform

All of our experiments were performed on our Rethink Robotics Baxter Research Robot, shown in Figure 5.1. It is equipped with two 7-dof arms, each with an electric parallel gripper. For the experiments in this paper, we use exclusively the right arm. The robot has an Asus Xtion Pro mounted on its upper-torso, directly below its screen, which includes both an RGB color camera and a depth sensor, each of which produce 640×480 images at 30Hz. Mounted on the robot immediately above the Xtion sensor is an Infrared Cameras Inc. 8640P Thermal Imaging Camera, which reads the temperature of the image at each pixel and outputs a 640×512 image at 30Hz.

5.2.2 Experimental Setup

For all experiments, the robot poured from the cup shown in its gripper in Figure 5.1. We used three target containers, also shown in Figure 5.1. We collected a dataset of pours using this setup in order to both train and evaluate our methodologies. We collected a total of 279 pouring sequences, in which the robot attempted to pour 250ml of water into the target using the thermal camera with the model-based method, with the initial amount in the cup varied between 300ml, 350ml, and 400ml. Each sequence lasted exactly 25 seconds and was recorded on both the thermal and RGBD cameras at 30Hz. We randomly divided the data

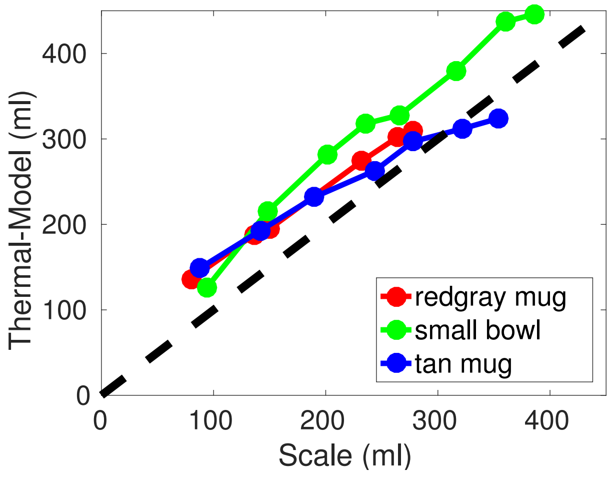
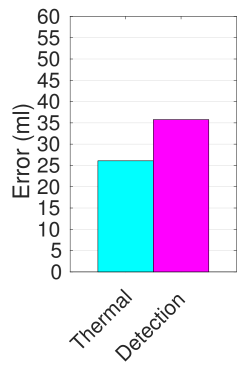


Figure 5.5: The plot shows the scale reading compared to the thermal camera with the model based method for each of the target containers.

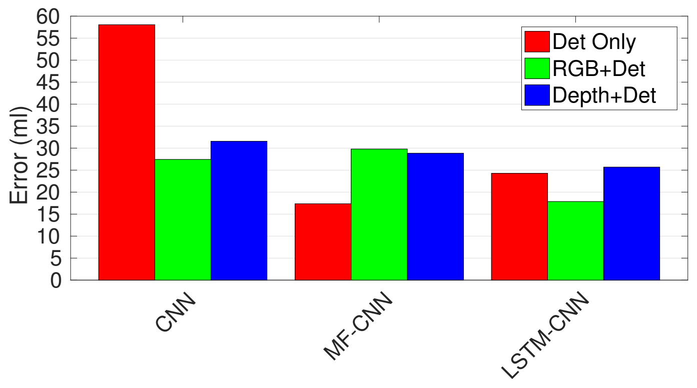
75%-25% into train and evaluation sets. After the data was collected, we used the thermal images to generate ground truth pixel labels as well as we used the Viterbi decoding method described in section 5.1.5 to generate ground truth volume estimates, which we compare against for the remainder of this section.

5.2.3 Validating Thermographic Ground Truth Methodology

Before we can evaluate our methodologies, we must first verify that our method for generating ground truth volume estimates is accurate. We can compare a static volume measurement with a scale to static estimates from the thermal camera combined with the model-based method to gauge the accuracy of our method. Figure 5.5 shows a comparison between measurements from a scale (x-axis) and the corresponding measurement from the thermal camera using the model-based method (y-axis) for each of the three target containers. The black dashed line shows a 1:1 correspondence for reference. From the figure it is clear that the model-based method overestimates the volume for each container. In order to make our



(a) Model-Based

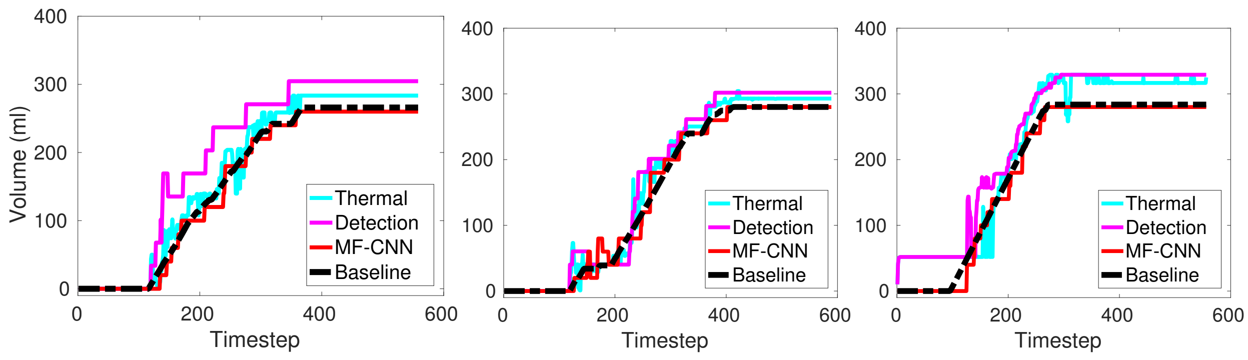


(b) Model-Free

Figure 5.6: Root mean squared error in milliliters of each of the methods for volume estimation. The left plot shows the error for the model-based method using either the thermal image or the output of the detection network as pixel labels. The right plot shows the error for the model-free methods when the networks take as input only the liquid detections (red), the liquid detections plus the color image (green), and the liquid detections plus the depth image (blue).

baseline as accurate as possible, we fit a linear model for each container and use that to calibrate the baseline ground truth estimates described in section 5.1.5.

Note that we only use this calibration for computing the ground truth baseline, and not when computing estimates for the model-based methodology. This is done on purpose, since a robot would not be able to pre-calibrate its estimates for every object in a household setting. While it might be reasonable to assume that a robot can acquire a 3D model for each target container via existing object databases, we believe that performing a pre-calibration using a scale and multiple pouring experiments for each object would be overly demanding.



(a) Small bowl (b) Tan mug (c) Redgray mug

Figure 5.7: The volume estimates for the two model-based methods and the multi-frame detection only model-free network. The black dashed line is the baseline ground truth. We randomly selected one sequence for each target container from our test set to display here. Best viewed in color.

5.2.4 Comparing Methods for Volume Estimation

For our model-free methodology, every network was trained using the mini-batch gradient descent method Adam [62] with a learning rate of 0.0001 and default momentum values. Each network was trained for 61,000 iterations, at which point performance tended to plateau. All single-frame networks were trained using a batch size of 32; all multi-frame networks with a window of 32 and batch size of 5; and all LSTM networks with a batch size of 5 and unrolled for 32 frames during training. The input to each network was a 160 $\times$ 160 resolution crop of either the liquid detections only, the color image and detections appended channel-wise, or the depth image and detections appended channel-wise. We discretize the output to 100 values for the range of 0 to 400ml (none of our experiments use volumes greater than 400ml) and train the network to classify the volume. The error signal was computed using the softmax with loss layer built into Caffe [58]. In our data we noticed that approximately $\frac{2}{3}$ of the time during each pouring sequence was spent either before or after pouring had occurred, with little change in the volume. We found that the best results could be achieved by first pre-training each network on data from the middle of each sequence during which the volume was actively changing, and then training on data sampled from the entire sequence. We discretize v_t and the output of the network into 20 values[6].

Figure 5.6 shows the root mean squared error in milliliters on the testing data for each method with respect to our baseline ground truth comparison described in section 5.1.5. It should be noted that although both our baseline ground truth estimate and the thermal estimate in Figure 5.6a are derived from the same data, the difference between the two can be largely attributed to the fact that the baseline method is able to look backwards in time and adjust its estimates, whereas the thermal model-based method can only look forward (which is necessary for control). For example, in the initial frames of a pour, as the water leaves the source container, it can splash against the side of the target container, causing the forward thermal estimate to incorrectly estimate a spike in the volume of liquid, whereas the baseline method can smooth this spike by propagating backwards in time.

While the error for both model-based methods are relatively small, it is clear that some of the model-free methods are actually better able to estimate the volume of liquid in the target container. Surprisingly, the best performing model-free estimation network is the multi-frame network that takes as input only the pixel-wise liquid detections from the detection network. The networks trained on detections only are the only networks that receive no shape information about the target container (both the depth and color images contain some information about shape), so intuitively, it would be expected that they would be unable to estimate the volume of more than a single container, and thus perform more poorly than the other networks. However, a lot of the temporal and perceptual information used by our methodology is already provided in the pixel-wise liquid detections, thus temporal information in addition to either color or depth images are not as beneficial to the networks.

We can verify that this is indeed the case by looking at the volume estimates on randomly

[6]While this may seem rather coarse, we found it works well in practice.

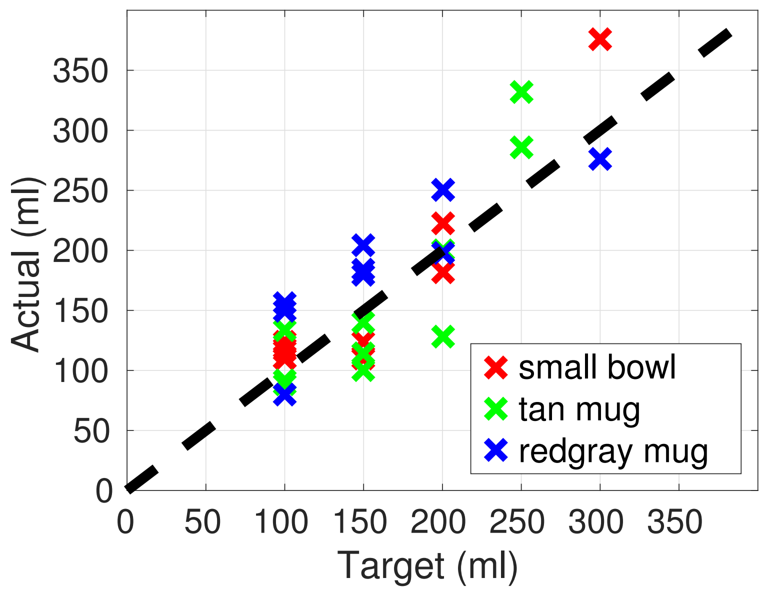
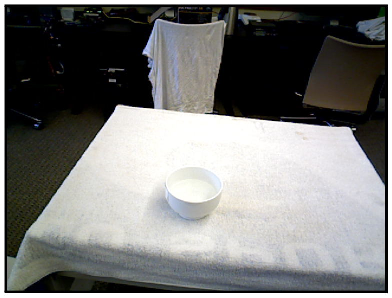
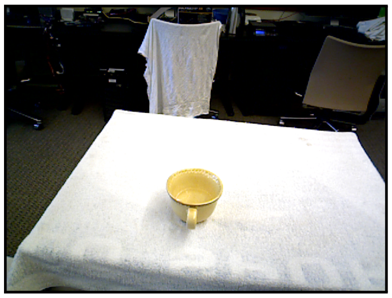
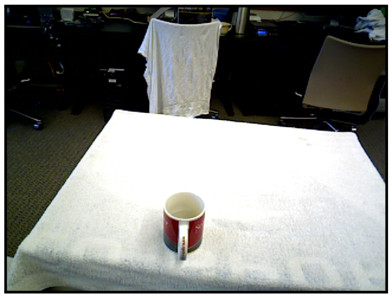


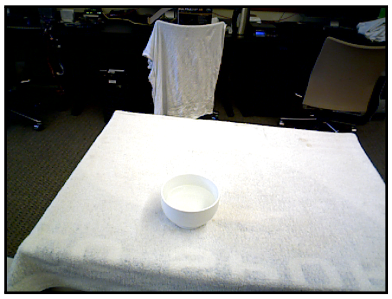
Figure 5.8: Plot of the result of each pour using our model-free method as input to the controller. The x-axis is the target amount that the robot was attempting to reach, and the y-axis is the actual amount the robot poured as measured by an external scale. The points are color-coded by the target container. The black dashed line shows a 1:1 correspondence for reference.

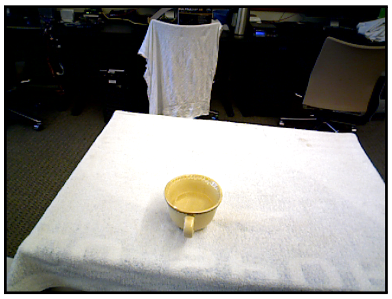
selected pouring sequences from the test set, one for each target container. Figure 5.7 shows the volume estimates for the two model-based methods and the multi-frame detection only method as compared to the baseline. It is clear from the plots that the multi-frame network is better able to match the baseline ground truth than either of the model-based methods. Not only does the multi-frame network outperform the model-based methods, but unlike them, it does not require either an expensive thermal camera or a model of the target container. For these reasons, we utilize this method in the next section for carrying out actual pouring experiments with closed-loop visual feedback.


(a) 200ml


(b) 100ml


(c) 150ml


(d) 250ml


(e) 150ml

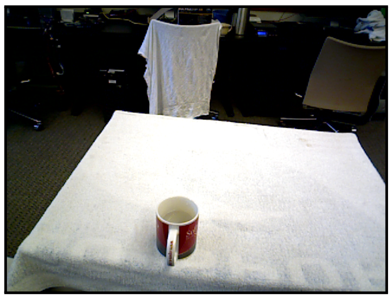

(f) 200ml

Figure 5.9: Reference images for each of the three target containers. This is exactly the perspective the robot sees when looking at the containers. Notice that a 50ml difference is difficult to perceive even for a human (best viewed enlarged).

5.2.5 Pouring with Raw Visual Feedback

Estimating the volume *a posteriori* and using a volume estimator as input to a pouring controller are two very different problems. A volume estimation method may work well analyzing the data after the pouring is finished, but that does not necessarily mean it is suitable for control. For example, if the estimator outputs an erroneous value at one timestep, it may be able to correct in the next since the trajectory of the pour does not change. However, if this happens during a pour and the estimator outputs an erroneous value, this may result in a negative feedback loop in which the trajectory deviates more and more from optimal, leading to more erroneous volume estimates, etc. To verify that our chosen method from the previous section is actually suitable for control, we need to execute it on a real robot for real-time control.

We test the best performing method from the previous section: the multi-frame network with detections from the detection network. To test the multi-frame network with detections only, we executed 30 pours on the real robot using the PID controller described in section 5.1.4. We ran 10 sequences on each of the three target containers. For each sequence, we randomly selected a target volume in $\{100, 150, 200, 250, 300\}$ milliliters and we randomly initialized the volume of water in the source container as either 300, 350, or 400 milliliters, always ensuring at least a 100ml difference between the starting amount in the source and the target amount (so the robot cannot simply dump out the entire source and call it a success). Each pour lasted exactly 25 seconds, and we evaluated the robot based on the actual amount of liquid in the target container (as measured by a scale) after the pour was finished.

Figure 5.8 shows a plot of each pour, where the x-axis is the target amount and the y-axis is the actual volume of liquid in the target container as measured by an external scale after the pour finished. Note that the robot performs approximately the same on all containers. This is particularly interesting since the volume estimation network is never given any information about the target container, and must simply infer it based on the motion of the liquid. Additionally, almost all of the 30 pours were within 50ml of the target. In fact, the average error over all the pours was 38ml. For reference, Figure 5.9 shows 50ml differences for each of our 3 containers from the robot's perspective. As is apparent from this figure, 50ml is a small amount, and a human solving the same task would be expected to have a similar error.

5.3 Conclusion and Future Work

In this chapter, we introduced a framework for visual closed-loop control for pouring specific amounts of liquid into a container. To provide real-time estimation of the amount of liquid poured so far, we developed a deep network structure that first detects the presence of water in individual pixels of color videos and then estimates the volume based on these detections. We then showed how the volume estimate could be given to a PID controller to perform real-time pouring of liquid.

Our experiments indicate that the deep network architecture can be trained to provide real-time estimates from color only data that are slightly better than the model-based estimates using thermal imagery. Furthermore, once trained on multiple containers, our volume estimator does not require a matched shape model of the target container. We incorporated our approach into a PID controller and found that it on average only missed the target amount by 38ml. While this is not accurate enough for some applications (e.g., some industrial settings), it is well suited for similar pouring tasks in standard home environments. The results here are important for the following chapters in this thesis. We showed that by using deep neural networks, a robot can robustly control a liquid even in the presence of noisy inputs to pour precise amounts of liquid. In the following chapter we investigate this perception model on a related task: *reasoning* about liquids.

5.3.1 Practical Takeaways

In this chapter we made many technical and practical insights as well as theoretical. The first was our use of a PD controller. In chapter 3 we performed the same task as in this chapter, albeit there we used guided policy search (GPS) to learn a control policy. However, here we used only a simple PD controller with no learned policy. Why the change? The biggest difference between chapter 3 and here is the delayed sensor measurements. There the delay in the scale reading made it impossible for a PD controller to solve the task consistently. When we began running the evaluations for this chapter, we initially began by training a policy using GPS. However, we quickly realized that the lack of sensor delay here meant that a PD controller with the right gains could solve the task fairly well. Given the complexity of the volume estimation pipeline, we decided to remove the potential extra confounding variables and use the PD controller in this chapter.

Another practical insight we had was on the issue of overfitting. We thoroughly analyzed this issue in chapter 4 and we found that the LSTM-FCN network with RGB input performed the best on *known* objects, but that it did tend to overfit to those objects. In this chapter we made the assumption that the robot would have had prior interactions with the objects, so we selected this network since it outperformed all others under these circumstances. However, we did still experience issues due to overfitting. For example, significant changes in lighting between the training data and when the robot was evaluated resulted in under performance of the network. Since this was the first step in the pouring pipeline, this would then cascade into the rest of the pipeline, degrading performance on the task. For the experiments in this chapter, we attempted to ensure similar conditions during evaluation as in the training set (e.g., similar lighting conditions), however it remains an open problem to see how performance can be maintained even when visual conditions of the environment may change.

Chapter 6

LEARNING TO SEE THROUGH OBJECTS: REASONING ABOUT OCCLUDED LIQUID

In this chapter, we examine a liquid reasoning task. We look at how a robot can locate the *unseen* liquid in a scene given the *visible* liquid. This is an important ability for a robot. Liquids are non-rigid, which makes it difficult if not impossible to directly manipulate them. Instead, in order to manipulate liquids, a tool or container must be used. This often has the undesirable side effect of occluding much of the liquid, making it difficult to perceive the full state of the liquid directly from sensors. Instead, the robot must rely on its knowledge of liquids to infer the locations of hidden liquid from observations, for example, by inferring the existence of liquid in a cup after observing the liquid's surface.

Here we examine the problem of *reasoning* about liquids, that is, we ask the question *Can the visible liquid be used to infer where all liquid is?* Specifically we look at the task of *tracking*. For *tracking*, we assume the robot has access to a perceptual system that can provide semantic labels for each pixel, like the one described in chapter 4 that can discriminate *visible* liquid from raw RGB images or like the ones in [87, 159] which provide labels for rigid objects. The robot's objective here is then to use that to find *all* liquid in the scene, both visible and occluded. This task requires the robot to reason about the relative relations between the various objects in the scene and the visible liquid in order to infer where hidden liquid is. For example, if the robot sees a group of pixels with the semantic label "cup" and a group of pixels with the label "liquid" exiting out the side of the cup, it can then infer that there must be liquid contained below that point in the cup. We use deep neural networks to allow the robot to learn these kinds of inferences directly from data.

[0]The contents of this chapter were published in [130] and [133].

This chapter is arranged differently from prior chapters. Unlike previous chapters, this chapter is not standalone; it builds significantly on chapter 4 and uses many of the same methods and datasets. We will describe the overview and differences in methodology here and refer the reader to that chapter for more details. The rest of this chapter is layed out as follows. The next section describes the task. The section after that describes the dataset used in the experiments in this chapter. The following section describes the learning methodology we use. The section after that details how we evaluate our methodology, and the following section contains our results. We finish this chapter with a discussion of the results and how they inform this thesis's investigation into robotic manipulation of liquids.

6.1 Task Overview

In this chapter we investigate the task of *reasoning* about liquids. We define *reasoning* to be, given semantic labels for the visible objects and liquid, determining where all the liquid is, even if it may not be directly perceivable (e.g., liquid inside a container). We call this task *tracking*. Specifically, we assume that the robot is given an image where each pixel in the image is labelled as the name of the visible object at that pixel (i.e., *liquid*, *cup*, etc.), and must output a binary label *liquid* or *not-liquid* for each pixel, where *liquid* labels also apply to liquid occluded by rigid objects. We assume the robot has access to a semantic labelling system in order to isolate the tasks of *reasoning* from the task of *perception*, however later in this chapter we also briefly examine combining the two together. We utilize deep neural networks to learn the inference of all liquid locations from semantic labels. For this chapter, we focus on the task of pouring as it requires reasoning about both where the visible liquid is as well as where hidden liquid is.

6.2 Data Set

We use the same simulated dataset as described in chapter 4 for all experiments in this chapter. Note that we do not use the real robot dataset from that chapter as the thermographic camera only gives labels for *visible* liquid and not *all* liquid, which means we cannot get the

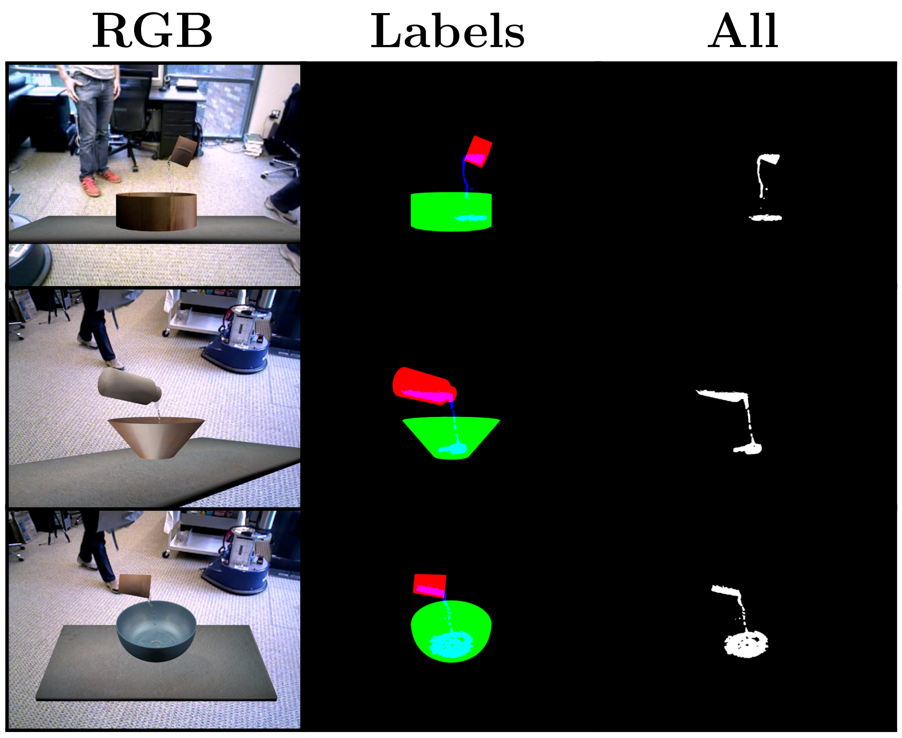


Figure 6.1: Examples of frames from the simulated dataset. The left column is the raw RGB images generated by the renderer (for reference); the center column shows the ground truth labeling output by the simulator; the right column shows the ground truth liquid location for all liquid in the scene.

ground truth to train against on the real robot. However, due to the known state in the simulator, we can "see-through" objects in order to get the ground truth on the simulated dataset. We refer the reader to chapter 4 for a full description of the simulated dataset. The rest of this section describes how we make the data generated for that dataset suitable for the task of *tracking*.

We generated the ground truth for each image in each rendered sequence as follows. For each object (source container, target container, and liquid), we set that object to render as a solid color irrespective of lighting (red, green, and blue respectively). Then we made all other objects in the scene invisible, and rendered the resulting scene. We then combine the images for the objects as separate channels of a single image (center column of Figure 6.1).

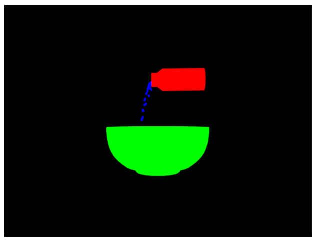
For the ground truth, we need the pixel labels for all the liquid, even the occluded liquid. The blue channel rendered in this way gives us the pixel labels we need. However, the input to the robot needs to be only the *visible* liquid as well as the labels for the topmost object. To do this, we render the scene again with each object rendered as its respective color, and then we encode which object is on top in the alpha channel of the ground truth image described in the last paragraph. Some examples of the result are shown in Figure 6.1. The left column shows the rendered color image for reference, the center column shows the ground truth pixel labels (absent the alpha channel), and the right column shows the liquid pixel labels for all liquid.

6.3 Learning Methodology

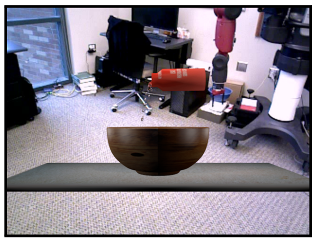
We use the same learning methodology as in chapter 4. Here we briefly summarize that methodology and we refer the reader to chapter 4 for more details. Our learning methodology utilizes deep neural networks to solve the *tracking* task.

We evaluate three different types of input formats for our network. The primary input type is the semantic label input. Each pixel of the input image is labelled with the type of object that is visible at that pixel. In the experiments in this chapter, there are 3 semantic labels: *cup* (for the source container), *bowl* (for the target container), and *liquid*. We imple-

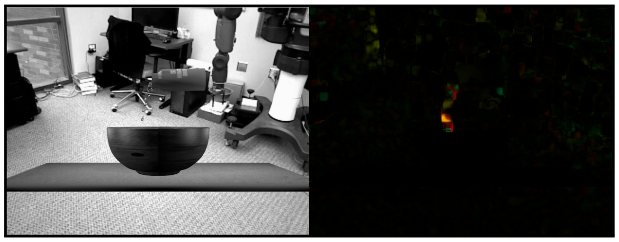
Inputs



(a) *Visible Objects*

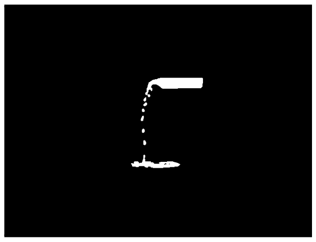


(b) *RGB*



(c) *Grayscale+Optical Flow*

Output



(d) *All Liquid*

Figure 6.2: Different images of the same frame from the same sequence. The upper part of this figure shows the different types of network inputs (RGB, semantic labels, and grayscale combined with optical flow). The lower part shows the type of desired network output (all liquid).

ment the labelling as 3 separate binary masks appended together channel-wise. Figure 6.2a shows an example with the red channel the label for the *cup*, the green channel the label for the *bowl*, and the blue channel the label for the *liquid*. Note that for each pixel, the vector of labels is one-hot, i.e., there is exactly one 1 and the rest are 0s, since only one object can be visible at once. We also perform an experiment where we combine the *detection* problem from chapter 4 with *tracking* into a single network. For that experiment, we evaluate both RGB (the default baseline input format) and grayscale combined with optical flow (the type with the best generalization performance in chapter 4) as inputs to the network. An example is shown in figures 6.2b and 6.2c.

For *tracking*, the desired output is the location of *all* liquid in the scene, including liquid occluded by the containers. Here the network must learn to infer where liquid is in the scene based on other clues, such as determining the level of liquid in the source container based on the stream of liquid that is visible coming from the opening. An example of this is shown in Figure 6.2d. The output the network is a pixel-wise label confidence image, i.e., for each pixel, the network outputs its confidence in $[0, 1]$ that that pixel is either *liquid* or *not-liquid*.

For the network architecture we use the same layouts as described in chapter 4. We evaluate all three network architectures: FCN, MF-FCN, and LSTM-FCN. The FCN is a standard fully convolutional network (FCN); it takes in an input image, passes it through a series of convolutional and non-linearity layers, and outputs pixel-wise labels. The second architecture, the multi-frame FCN (MF-FCN), is similar to the FCN, with one major difference. It takes in a series of frames in sequence and passes each individually through a set of convolutional and non-linearity layers, then combines the output channel-wise and passes the combination through a final set of layers before producing pixel-wise labels. The advantage of this layout over the standard FCN is that it is able to see movement via the sequence of images, potentially making it easier to reason about liquids. The final architecture we evaluate is the LSTM-FCN. This architecure replaces one of the final convolutional layers of the FCN with a long short-term memory (LSTM) layer [54]. This allows the network to have an explicit memory that it can pass forward from timestep to timestep. Additionally, we add the network's output from the previous timestep as an input, again giving it the ability to "remember" from one frame to the next. Figure 4.10 in chapter 4 shows each of these networks and describes them more in detail.

One final difference between the methodology in this chapter and that in chapter 4 is the use of input blocks. In that chapter, part of out evaluation was whether or not to combine input types and if so to use early or late fusion. However, here our focus is on the *tracking* task and so for both the semantic label input and RGB we use the standard input block, and for grayscale with optical flow input we use the early-fusion input block since it had the best performance in chapter 4.

6.4 Evaluation

We evaluate all three of our network types on the task of *tracking* on the simulated dataset. We report the results as precision and recall curves, that is, for every value between 0 and 1, we threshold the confidence of the network's labels and compute the corresponding precision and recall based on the pixel-wise accuracy. We also report the area-under-curve score for the precision and recall curves. Additionally, we report precision and recall curves for various amounts of "slack," i.e., we count a positive classification as correct if it is within n pixels of a true positive pixel, where n is the slack value. This slack evaluation allows us to differentiate networks that are able to track the liquid, albeit somewhat imprecisely, versus networks that fire on parts of the image not close to liquid.

In chapter 4 we evaluated the network on two subsets of the data: the *fixed-view* set and the *multi-view* set. The *fixed-view* set contains all the data for which the camera was directly across from the table (camera azimuth of 0 or 180 degrees) and the camera was level with the table (low camera height), or 1,266 of the pouring sequences. Due to the cylindrical shape of all the source and target containers, this is the set of data for which the mapping from the full 3D state of the simulator to a 2D representation is straightforward, which is useful for our networks as they operate only on 2D images. The *multi-view* set contains *all* the simulated camera viewpoints, rather than just the ones directly facing the objects.

The mapping from 3D to 2D for this set is not as straightforward and thus we focus our evaluation in this chapter for *tracking* on just the *fixed-view* set. This is a limitation of our 2D reasoning methodology which we will address in future chapters.

For the task of *tracking*, we trained the networks on segmented object labels (Figure 6.2a). That is, assuming we already have good detectors for what is visible in the scene, can the robot find the liquid that is not visible? Note that here we use the ground truth labels as shown in Figure 6.2a and not the output of the detection network as input to the tracking network, however we do also evaluate combining the two. Since the input image is already somewhat structured, we scale it down to 130×100 pixel resolution. Unlike for *detection* in chapter 4, here we don't pre-train the networks on crops, but we do utilize the same gradient weighting scheme, i.e., we weight the loss on negative (*not-liquid*) pixels by 0.1 to emphasize the loss on positive (*liquid*) pixels. This is to overcome the imbalance in the dataset as the vast majority of pixels do not contain liquid.

We used the Caffe deep learning library to implement all our networks [58]. We trained each network for 61,000 iterations. The single-frame networks were trained with a batch size of 32. The multi-frame networks were given a window of 32 frames as input and were trained with a batch size of 1. The LSTM networks were unrolled for 160 frames during training (i.e., the gradients were propagated back 160 timesteps) and were trained with a batch size of 5. We used the mini-batch gradient descent method Adam [62] with a learning rate of 0.0001 and default momentum values. All error signals were computed using the softmax with loss layer built-into Caffe [58].

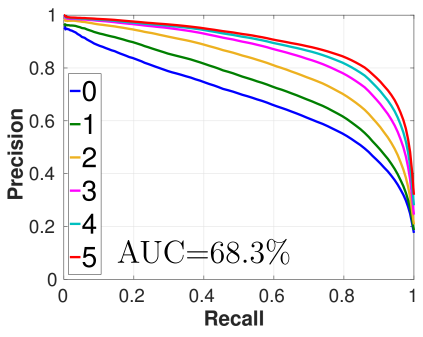
Finally, we also evaluated the performance of combined *detection* (from chapter 4) & *tracking* with a single network. The networks take in the same 400×300 images that the *detection* networks take, and output the location of *all* liquid in the scene. We initialize these networks with the weights of their corresponding *detection* network and train them on full images. We use the same gradient weighting scheme as for the two tasks separately. We train the networks for combined *detection* & *tracking* using the same learning parameters as for training the *detection* networks as described in chapter 4.

6.5 Results

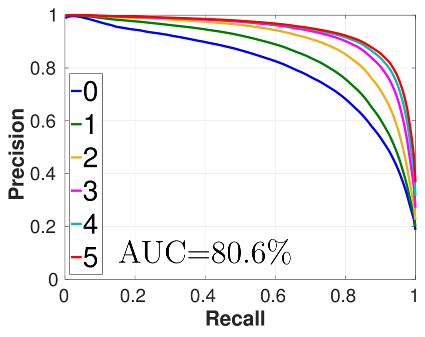
Figure 6.3 shows the performance of the 3 network types on the *tracking* task. As expected, the only network with an explicit memory, the LSTM-FCN, performs the best. However, the other two networks perform better than would be expected of networks with no memory capability. This is due to the fact that, given segmented input, the networks can infer where some of the liquid *likely* is. Although it is clear that LSTM-FCNs are best suited for this task. We additionally tested the LSTM-FCN on the combined *detection* & *tracking* task since it had the best performance on each task individually. The results when using the standard RGB format as input are shown in Figure 6.3d. The network in this case is able to do quite well, with only a minor drop in performance as compared to the LSTM-FCN on the *tracking* task alone.

We also wish to compare the best performer on the *detection* task with the default baseline of RGB input. In chapter 4 we found that to be grayscale with optical flow combined using the early-fusion input block. Chapter 4 showed that an LSTM-FCN taking as input grayscale early-fused with optical flow has the ability to generalize better than any other type of network we evaluated. This result was achieved on the *detection* task, and we wanted to see if this translates to the *tracking* task. However, since the robot data set used in that chapter does not contain the ground truth for tracking, we use the simulated dataset to test this hypothesis.

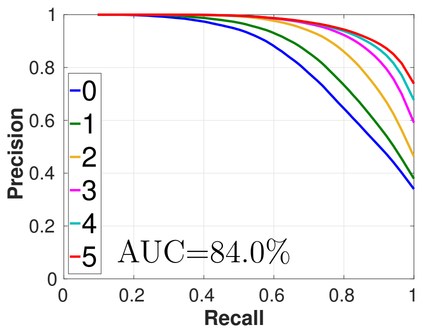
We train two networks: one that takes the default input of RGB images and one that takes grayscale images early-fused with optical flow. We train them on the combined *detection* & *tracking* task. They are trained in the same manner as described previously for doing combined *detection* & *tracking*. The advantage of using this alternative input type is its ability to generalize to new data, so we hold out all pouring sequences with one of the target containers (the *dog dish*) during training. This includes during training of all pre-trained networks such that the final weights of the networks were never influenced by any data containing the test object.



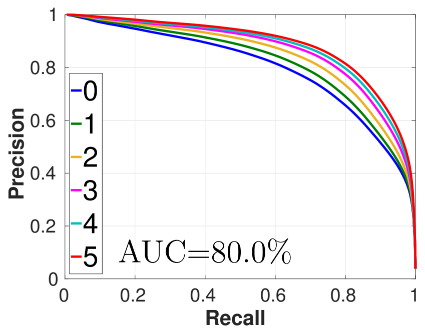
(a) FCN



(b) MF-FCN

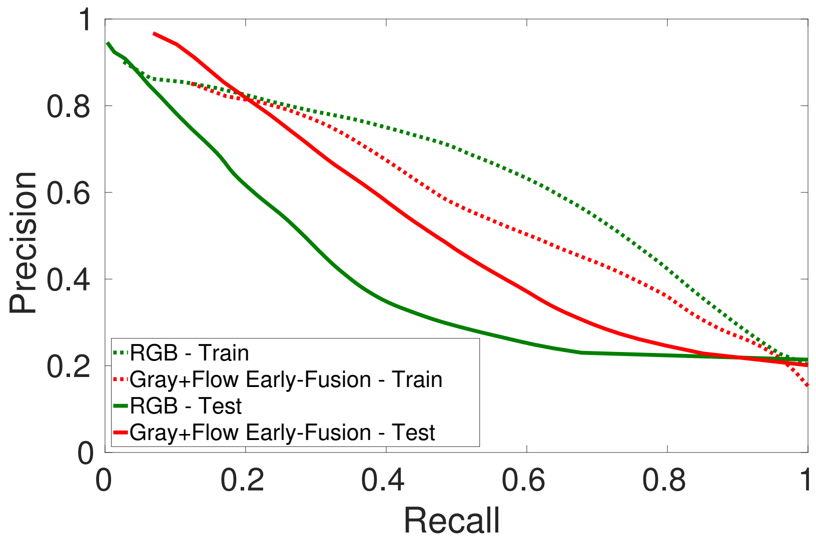


(c) LSTM-FCN



(d) Combined LSTM-FCN

Figure 6.3: The precision-recall curves for the *tracking* task on the simulated dataset. The first 3 show the performance of the three network types on the *tracking* task alone. The last graph shows the performance of the LSTM-FCN on the combined *detection* & *tracking* task using the standard RGB format as input. The different lines show the different amounts of slack, i.e., how far a positive classification can be from a true positive to still count as correct. The area under the curve (AUC) is shown for the 0 slack curve.



(a) Precision-Recall

	Train	Test
RGB	63.6%	40.1%
Gray+Flow Early-Fusion	57.6%	52.2%

(b) AUC

Figure 6.4: The performance on the combined *detection & tracking* task of the LSTM-FCN that takes as input RGB images compared to the performance of the LSTM-FCN that takes as input grayscale images plus optical flow using the early-fusion input block. The upper graph shows the precision-recall plot for both networks on both the train and test sets. The lower table shows the corresponding area under the curve for each curve. Note that here for clarity we don't use any slack unlike in figures 6.3 (equivalent to a slack of 0).

Figure 6.4 shows the performance of the two networks on both the train and test sets. From this figure it is clear that the RGB network outperforms the other on the training set, however, the gray+flow early-fusion network outperforms the RGB network on the test set. This confirms the results we found in the chapter 4: Networks trained with grayscale early-fused with optical flow generalize better to new situations for tasks dealing with raw perceptual input.

6.6 Discussion

In this chapter we evaluated a robot's ability to *reason* about liquids via the *tracking* task. In this task, the robot was required to determine the locations of all liquid in a scene (visible and occluded) given only semantic labels of the visible objects. We evaluated the same network architectures as in chapter 4: FCN, MF-FCN, and LSTM-FCN. As in that chapter, we found the best performance was by the LSTM-FCN, the only network with an explicit memory mechanism. We also evaluated our networks on combining the *detection* task from chapter 4 with the *tracking* task. Once again, the LSTM-FCN had the best performance, reinforcing our conclusion that in order to effectively reason about liquids, the robot must be able to integrate information over time.

This chapter revealed several useful insights about a robot's ability to reason about liquids. First of all, it is possible to learn basic, intuitive properties of liquids from data, such as if there is liquid pouring from the lip of a cup then there must be liquid filling the cup below the lip, if liquid disappears from sight into a container it remains in the container even if unseen, or if liquid is unsupported it will fall. Also, good semantic labels can enable the robot to robustly reason about the behavior of liquid in its environment. And finally, when reasoning about liquids some notion of memory or state passed forward from frame to frame can enable the robot to reason more robustly, rather than reconstructing it's knowledge at every timestep.

However this chapter also reveals one major flaw in our methodology so far. We only evaluated our networks on the *fixed-view* set because that set viewed the objects directly from the side, creating an almost one-to-one correspondence between the 2D pixel labels and the true 3D liquid state. This is not always the case, as exemplified by the *multi-view* set, which we excluded from our analysis for this reason. In real environments, the true state of liquid is 3D, but our image-based methodology can only reason in 2D. If we really want robots to robustly reason about liquids, they'll need full 3D reasoning capabilities. In deep learning there has not been very much work on combining 3D fluid dynamics with deep neural networks (what work there is we discussed in chapter 2). However there is a significant amount of work in the fields of fluid dynamics and computational graphics on handling liquids in 3D. In the next part of the thesis we switch tracks and examine how we can use this work to enable robots to reason about liquids in 3D.

6.6.1 Practical Takeaways

In the process of executing the research in this chapter, we had to make several methodological choices due to practical or technical reasons. The first was the use of the simulated dataset from chapter 4 for training and evaluation, and not the real dataset also generated in the same chapter. Ideally, we would like to evaluate on data collected in a real environment. However, unlike for a simulated dataset, "seeing-through" the sides of real objects to acquire ground truth labels is more challenging. One possibility is to use a container made of a transparent material such as glass. But while glass allows visible light to pass through, infrared light, the light used by our thermal camera to acquire ground truth labels, is blocked. An alternative is to use a container made of a material that is transparent to infrared light such as silicon, which would allow us to generate ground truth labels for the *tracking* task but also disallow the robot from seeing into the container with RGB (unlike glass). For this work we attempted to locate a container made of silicon for precisely this purpose, however modern kitchen containers advertised as made of silicon are actually made of silicon combined with another material, resulting in them being opaque to infrared light. Thus if we wanted a pure silicon container, we would have had to make it ourselves, which from a practical standpoint is prohibitively expensive. So for this chapter, we focused our evaluation on where it was feasible, the simulated dataset.

Another methodology choice we made was the design of the LSTM-FCN. For the task in this chapter, *tracking*, the ability to remember information from frame to frame is important, even more so than for the *detection* task from chapter 4. So the fact that only the LSTM-FCN has an explicit memory seems odd. Why not give the FCN and MF-FCN access to their own output from the previous timestep so they can also have an explicit memory? Imagine that we do this, we alter the FCN such that it also takes its own output from the previous timestep. During training, we need to feed something for this input into the network. We could use the ground truth pixel labels, but if we do this the network will become too reliant on having very accurate and precise labels from the prior timestep so that when we evaluate it and it inevitably deviates, it will be unable to correct because it assumes the previous labels are highly accurate. We could instead feed in all zeros, however in this case the network would simply learn to ignore the previous labels. The best way to do this is to train the network to correct for its own mistakes by unrolling it in time (the same way we train the LSTM-FCN), thus passing its own actual output to itself during training. The issue with this is that once the network is unrolled for more than a few timesteps, it runs into the exploding/vanishing gradient problem (that is, there are enough multiplications in a row during the backward pass to cause the gradients to overflow/underflow the amount that can be stored in a floating point value). To prevent this, we need to insert an LSTM layer, which is explicitly designed to handle this problem, into the network. The result of modifying the FCN in this way is exactly the LSTM-FCN. This is in fact the line of reasoning we followed to create the LSTM-FCN in the first place.

Part II

INCORPORATING FLUID PHYSICS MODELS

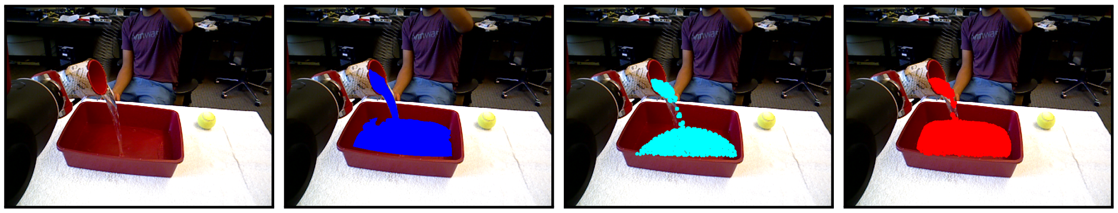
Chapter 7

FROM 2D TO 3D: REASONING ABOUT LIQUIDS USING LIQUID SIMULATORS

In the previous chapters, we investigated using deep convolutional neural networks to enable robotic interaction with liquids. That work focused largely on model-free learning-based methodologies. However, as we noted in the last chapter, the biggest limitation to that is its reliance on 2D images for reasoning. Real liquid state is in 3D and there is not always a straightforward mapping to 2D for every task (as was the case for the *multi-view* set). So in this chapter, we make the leap from 2D to 3D reasoning. Research on interfacing 3D data with deep networks is somewhat limited, especially as it concerns liquids, so we start by applying known methods to this task. We utilize physics-based fluid dynamics models to represent the 3D state of the liquid and perform reasoning tasks.

Physics-based models are very general tools for enabling robots to reason about their environments. Work on rigid-body actions using physics-based models has enabled robots to perform a wide variety of tasks [145, 117, 21]. However, to use such models requires tracking their state using real-time perception. For rigid-body models and deformable objects such as clothing and towels, there has been a lot of work on tracking the modeled state using sensory feedback [97, 135, 137]. For liquids, though, there has not yet been any work connecting physics simulation with real-time perception for robotic tasks. Unlike modeling rigid or deformable bodies, modeling liquids is much higher dimensional and lacks the same kind of inherent structure, thus small perturbations can quickly lead to large deviations. As an example, Figure 7.1 shows a comparison between real liquid (Figure 7.1b) and the result of performing a carefully tuned liquid simulation with the same setup (Figure 7.1c). It is

[0]The contents of this chapter were published in [131].



(a) Color image (b) Ground Truth (c) Open-loop sim (d) Closed-loop sim

Figure 7.1: A comparison between open-loop and closed-loop liquid modeling. The left-most figure shows the color image of the scene for reference and the left-center figure shows the same image with the actual liquid pixels labeled. The right two images show the color image, but with the liquid from the simulator shown.

clear that without any feedback, the liquid simulator and the real liquid have significant differences.

In this chapter, we investigate ways to incorporate sensory feedback into physics-based liquid simulation. By closing the loop between simulation and real-time observations, a robot can track liquids with much higher accuracy, as illustrated in Figure 7.1d. Ultimately, the ability to accurately track the state of a liquid will enable a robot to reason about liquids in a wide variety of contexts, addressing questions such as “How much water is in this container?”, “Where did this liquid come from?”, “What is the viscosity of this liquid?”, or “How can I move a specific amount of this liquid without spilling?”. Toward this goal, our work only assumes that the robot can track 3D mesh models of the objects in its environment and can differentiate between liquid and everything else in its camera observations, both tasks that have been addressed in prior work [136, 35]. We demonstrate that our closed-loop liquid simulation enables a robot to reason about liquids in ways that were infeasible before, such as estimating the amount of water in an opaque container during a pouring task, or detecting partial obstruction in water pipes.

The rest of this chapter is laid out as follows. We start in the next section by giving a detailed description of the liquid simulator we use as the base for our closed-loop physics-based model. After that we describe two different methods for using the observations of real liquid to correct errors in the base liquid simulator. Next we describe three experiments we performed using this methodology and their results. We end the chapter with a discussion of the implications of our method and how it informs our investigation of liquid manipulation.

7.1 Open-Loop Liquid Simulator

Our physics-based model is based on a liquid simulator. The state of the simulator tracks the liquid over time, simulating it forward while observations prevent it from deviating from the real liquid dynamics. In this section, we describe how the liquid simulator computes the dynamics of the liquid, and in the following section we describe how the observation modifies the liquid state.

To simulate the trajectory of liquid in a scene, the liquid is represented as a set of particles and the Navier-Stokes equations [2] are applied to compute the forces on each particle. The Navier-Stokes equations require certain physical properties of liquid (e.g., pressure, density) to be defined for all points in $\mathbb{R}^3$. This is implemented using Smoothed Particle Hydrodynamics (SPH) [155], which computes the value of a property at a specific location in space as the weighted average of the neighboring particles. This is in contrast to finite element liquid simulations [122], which divide the scene into a voxel grid and store the values of the given property at each location in the grid. One major disadvantage of the finite element simulations is that as the size of the environment grows, the requirements of the voxel grid in both memory and run time grows as $O(n^3)$, making them inefficient for large environments with sparsely located liquids. This is the case for the simulations in this chapter and thesis as well, and so we chose to use SPH, which is better suited to this type of task. The implementation used in this chapter is based off the implementation from the particle simulation library Fluidix [89]. The rest of this section briefly describes that implementation.

Smoothed Particle Hydrodynamics is essentially a method for representing a continuous vector field of a physical property in space via a discrete set of particles. It is based around the following equation for evaluating that field at any arbitrary point in space, where A is the physical property in question:

$$A(r) = \sum_j m_j \frac{A_j}{\rho_j} W(|r - r_j|, h)$$

where m_j is the mass of particle j, A_j is the value stored in particle j, ρ_j is the density of particle j, W is a kernel function that weights the contribution of each particle by its distance, and h is the cutoff distance for W. In SPH, the mass m_j of each particle is constant, however the density ρ_j is not, and must be computed via the SPH equation above. That is, the physical value we want to compute A is set to be the density ρ, which results in ρ appearing on the right side of the equation twice. The issue of recurrence (requiring the density to be known in order to compute the density) is handled by the density in the denominator canceling out:

$$\rho(r) = \sum_j m_j \frac{\rho_j}{\rho_j} W(|r - r_j|, h) = \sum_j m_j W(|r - r_j|, h).$$

To implement a liquid simulation using SPH, each particle must store 6 physical properties: 3D position (without orientation however since particles are infinitesimally small points), velocity, force, mass, density, and pressure. As stated above, the mass for each particle is constant. At each timestep, the force is used to update the velocity as follows:

$$v_i^{t+1} = v_i^t + \frac{f_i^t}{\rho_i} \Delta T$$

where ΔT is the amount of simulation time one timestep corresponds to. The position at each timestep is then updated by the velocity in a similar manner

$$r_i^{t+1} = r_i^t + v_i^t \Delta T.$$

The density of each particle at each timestep is computed using the equation in the previous paragraph. The pressure is computed as

$$p_i = c_i^2(\rho_i - \rho_0)$$

where c_i^2 is the speed of sound and ρ_0 is the reference density of the liquid.

The force is computed by summing the contributions from pressure, viscosity, gravity, and surface tension. The pressure force at particle i is defined as:

$$f_i^{pressure} = \sum_j -\frac{m_j}{\rho_j}\left(\frac{p_i}{\rho_i^2} + \frac{p_j}{\rho_j^2}\right)\nabla W(r_i - r_j).$$

The force due to viscosity is

$$f_i^{viscosity} = \sum_j -\mu\frac{m_j}{\rho_j}\left(\frac{v_i}{\rho_i^2} + \frac{v_j}{\rho_j^2}\right)\nabla^2 W(r_i - r_j)$$

where μ is the viscosity constant of the liquid (recall that v_i is the velocity of particle i). To compute the surface tension acting on each particle, we must first compute the normal of each particle:

$$n_i = \sum_j \frac{m_j}{\rho_j}\nabla W(r_i - r_j).$$

Intuitively, the normal n_i for any particle in the center away from the surface of the liquid will have approximately equal contributions from all directions, resulting in the magnitude of n_i being small. Conversely, for particles near the surface, n_i will have a large contribution from particles in the direction of the interior of the liquid and very little contribution in the direction of the surface, resulting in an n_i with a large magnitude in the direction away from the surface. The force due to surface tension is computed as

$$f_i^{tension} = -\sigma\frac{n_i}{|n_i|}\sum_j \frac{m_j}{\rho_j}\nabla^2 W(r_i - r_j)$$

where σ is the liquid's tension constant. To prevent numerical instability when $|n_i|$ is small, we only compute the tension force when the normal magnitude is greater than a threshold, i.e., the particle is near the surface.

To simulate the flow of liquid in a scene during an interaction, we assume the simulator is given 3D models of the objects that interact with the liquid as well as their 6D poses over the course of the interaction (obtained for example from an object tracking system such as [136]). We initialize the liquid particles in the scene (details on this in section 7.4) and simulate the particles forward at each timestep as the simulator tracks the objects' poses.

Our liquid simulator is implemented using the particle simulation library Fluidix [89], which efficiently computes particle interactions on the GPU. We performed a best-first grid search over the space of parameters (e.g., the viscosity constant) to find the set of values that best match the real liquid dynamics. For each set of parameters in the grid, we used the evaluation criteria described in section 7.3.4 to score them with respect to the data we collected (described in section 7.3.2), and selected the parameters that best fit the real data. In doing so, we attempted to make our open-loop simulation as close as possible to the real liquid dynamics. For efficiency reasons, we use between 2,000 and 8,000 particles in our experiments. For a detailed derivation of Smoothed Particle Hydrodynamics, please refer to [155].

7.2 Closed-Loop Liquid Simulators

While liquid simulators model fluid dynamics based on physical properties, they often don't model every possible force that could affect the liquid; and even the best simulators still have some error relative to real liquids. Over time, even small errors can lead to a large divergence between real and simulated liquid behavior. While this may not be a problem in some cases (e.g., in 3D animation it may only be necessary for a liquid to appear realistic), if we wish to use liquid simulation as a robot's internal model of its environment, it must match the real liquid behavior as closely as possible.

One potential method for alleviating this issue is to improve the fidelity of the simulator. However, this method has many pitfalls. It requires knowledge of every possible force that could affect the trajectory of the liquid, not only the standard forces such as pressure and viscosity, but also forces for example due to vacuum suction (as in the case of a plunger), which may require modeling additional elements of the environment. It can also be very brittle, as every property of every object in the environment must be known ahead of time (e.g., the friction constants over the entire surface of every object). Finally, and most importantly, even if the simulator is almost perfectly accurate, the initial state of the simulator might not be known (e.g., unknown amount of water in a cup), and it will still deviate slightly from reality and thus accumulate drift, which a purely open-loop system has no way to estimate or correct for.

We propose two methods for dealing with noise when tracking real liquid dynamics using a simulator. Both methods involve closing the loop, that is, utilizing observations of real liquid dynamics in order to better approximate them in the simulation. The first, inspired by standard Bayes filters in robotics, is a MAP filter, which uses the observation to "correct" simulation errors relative to the observation. The second, based on modeling physical forces in the simulator, applies a warp field that pulls particles toward observed liquid. We describe these two methods in the following sections.

7.2.1 Bridging the Observation and the State

Before describing our two closed-loop methods, we briefly describe how we map the full 3D state of the liquid simulator into the robot's perception space. In this chapter, we assume that the robot's camera only provides 2D images labeled with pixel detections, based on the observation that most liquids, especially water, are not detected by depth cameras (this is the same as the output of our networks in chapter 4). At any timestep t, the robot's perception is thus a binary image I_t, with pixels labeled as *liquid* or *not-liquid*. In order to directly compare the particles representing the 3D liquid state with the 2D image, the pose of the particles must be projected into the image. This is done using the following equation:

$$\widehat{x}_t^i = A x_t^i \left(\begin{bmatrix} 0 & 0 & 1 \end{bmatrix} x_t^i \right)^{-1}$$

where x_t^i is the pose of particle i at time t, $\widehat{x}_t^i$ is that pose projected onto the 2D image plane, and A is the camera intrinsics matrix:

$$\begin{bmatrix} FL_x & 0 & PP_x \\ 0 & FL_y & PP_y \end{bmatrix}$$

where FL is the focal length and PP is the principle point of the camera. When projecting particles into the image plane, we can take into account occlusions by casting a ray from the particle's 3D pose into the camera's 3D pose and checking if it collides with any of the rigid objects in the scene. Any particle whose ray collides with an object is not included when updating the dynamics of the simulator as there is no way to directly observe that particle. For the particles that are not occluded, we can compute the distance in 2D space between pixels in the image and liquid particles, which can then be used to inform the dynamics of the liquid simulator.

Additionally, we can use this projection to compute the likelihood of an image, that is, how well the overall set of liquid particles "explains" each of the observed pixels. We define the function Φ to be the *coverage* function that maps a pixel location to the number of particles that cover that pixel. To compute this, we place a small, fixed radius sphere at each liquid particle location, then project those spheres back into the camera, ignoring occluded spheres. The value of Φ at a given pixel location is then simply the number of these spheres that projected onto that pixel. We use this function in both our closed-loop methods.

7.2.2 MAP Filter Simulator

We use a maximum *a posteri* (MAP) filter as one of our closed-loop simulation methods. We model each particle as its own filter, with its own set of hypotheses, and use the MAP hypothesis at each time step to compute the dynamics. Let $\mathcal{P}_t$ be a set of liquid particles in a scene at time t, $\mathcal{O}_t$ be the objects and their corresponding 6D poses, and I_t be the observation. We define $\mathcal{S}\left(\mathcal{P}_{t-1}, \mathcal{O}_t\right) = \mathcal{P}_t$ to be the function as described in section 7.1 that computes the state of the liquid particles at timestep t given the previous state of the liquid particles.

At the beginning of each timestep t, all the liquid particles are propagated forward in time by one step via $\mathcal{S}$ using the objects and their poses $\mathcal{O}_t$. Since S is deterministic, we perform the dynamics sampling step in the filter separately. Given a liquid particle x_t^i, we sample one hypothesis particle $\tilde{x}_t^{i,n}$ for each location in a grid centered at that liquid particle's position. The grid has dimension $3 \times 3 \times 3$ and the size of each grid cell is set at a small, fixed constant (we use 5mm in this chapter). This results in 27 hypotheses sampled for each liquid particle.

Next we must compute $P(\tilde{x}_t^{i,n}|I_t, \mathcal{P}_t)$, the probability of each hypothesis particle given the observation and the set of liquid particles. Here, we must condition on all particles in order to take into account that these particles may already "explain" a certain liquid pixel. We first apply Bayes rule

$$P(\tilde{x}_t^{i,n}|I_t, \mathcal{P}_t) \propto P(I_t|\tilde{x}_t^{i,n}, \mathcal{P}_t)P(\tilde{x}_t^{i,n}|\mathcal{P}_t).$$

For simplicity, we use a uniform prior $P(\tilde{x}_t^{i,n}|\mathcal{P}_t)$ over all hypothesis particles that are feasible, eliminating those that violate physical constraints, such as moving through a 3D object mesh. Thus, for all feasible hypothesis particles,

$$P(\tilde{x}_t^{i,n}|I_t, \mathcal{P}_t) \propto P(I_t|\tilde{x}_t^{i,n}, \mathcal{P}_t).$$

When computing $P(I_t|\tilde{x}_t^{i,n}, \mathcal{P}_t)$, what we really want to know, since this is a MAP filter, is which $\tilde{x}_t^{i,n}$ maximizes this probability. However, the interaction between I_t, $\tilde{x}_t^{i,n}$, and $\mathcal{P}_t$ is highly complex and difficult to compute analytically. Instead, we approximate this value with an activation function Ψ which we define to be

$$\Psi(I_t, \tilde{x}_t^{i,n}, \mathcal{P}_t) = \sum_{j \in liquid(I_t)} \frac{W(|\widehat{\tilde{x}}_t^{i,n} - j_t|, h)}{\Phi(j_t, \mathcal{P}_t) + 1}$$

where $liquid(I_t)$ is the set of all *liquid* pixels in I_t, W is a kernel function, $\widehat{\tilde{x}}_t^{i,n}$ is $\tilde{x}_t^{i,n}$ projected onto the image plane (as described in the previous section), h is the limiting radius for W, and Φ returns the coverage of j_t by $\mathcal{P}_t$ (also described in the previous section). Intuitively, this function sums the number of *liquid* pixels around $\tilde{x}_t^{i,n}$, weighted by their distance to $\widehat{\tilde{x}}_t^{i,n}$ divided by their coverage, i.e., how well explained that pixel is by $\mathcal{P}_t$. Thus, the more *liquid* pixels around a hypothesis particle, the higher its Ψ value, and the less the pixels are covered by the liquid particles, the higher the Ψ value. For W we use a squared exponential kernel with a length scale of $\frac{1}{33^2}$, and we set the limiting radius to 100. Intuitively, this means that the unit length under this kernel is 33 pixels with a limiting radius of 100 pixels.

Finally, we set x_t^i from the MAP hypothesis particles as follows:

$$x_t^i = \underset{\tilde{x}_t^{i,n}}{\text{argmax}}\ \Psi(I_t, \tilde{x}_t^{i,n}, \mathcal{P}_t).$$

Note that we also adjust the velocity of x_t^i to match the change in position from x_{t-1}^i so as to preserve the correct momentum.

7.2.3 Warp Field Simulator

The second method we use for closing the loop in the simulator is a warp field, somewhat similar to the approach applied in [137]. Here, the observation applies a force in the simulator that attempts to make the liquid particles better match the observed liquid. Each observation point is essentially a magnet in the scene, pulling nearby particles towards it. However, if all observation points pulled with the same amount of force, then particles would tend to clump around a subset of the observation points, leaving other observation points with no nearby particles as the forces from the former cancel out those from the latter. Thus, the amount of force an observation point applies to nearby particles must vary with the number of nearby particles. When taken together, all the observation points create a field of forces that warp the particles to better match the real liquid observations.

Once again let $\mathcal{P}_t$ be a set of liquid particles in a scene at time t, $\mathcal{O}_t$ be the objects and their corresponding 6D poses, I_t be the observation, and $\mathcal{S}$ be the function that computes the dynamics of the particles for a single timestep. The force due to the observation warp field is computed as

$$\widehat{f}_t^{i,obs} = \sum_{j \in liquid(I_t)} \lambda \frac{u_t^{ij}}{\Phi(j_t, \mathcal{P}_t) + 1} W(|\widehat{x}_t^i - j_t|, h)$$

where λ is the warp constant, $liquid(I_t)$ is the set of all *liquid* pixels in I_t, u_t^{ij} is a unit vector pointing from particle $\widehat{x}_t^i$ (projected onto the image plane as described in section 7.2.1) to liquid pixel j_t, $\Phi(j_t, \mathcal{P}_t)$ is the coverage of pixel j_t (described in section 7.2.1) and W is the same kernel function used in the MAP simulator (with same parameters). The warp constant λ adjusts the strength of the warp force, with higher values resulting in a higher warp force and lower values in a lower force.

Again, the coverage of a pixel $\Phi(j_t, \mathcal{P}_t)$ is a measure of how many liquid particles "cover" it, that is, how many liquid particles are nearby. The force applied to each particle by each liquid pixel is divided by that pixel's coverage, thus as more particles cover an observed liquid pixel, it pulls particles to it with less force. Conversely, pixels that have lower coverage pull particles to them with more force, thus encouraging the simulator to move particles so as to fill the contour of the observed liquid.

The force $\widehat{f}_t^{i,obs}$ is then projected back into 3D space. This is done by applying the inverse of the projection described in section 7.2.1. Because this is 2D to 3D, the projection has an unspecified degree of freedom. To compensate for this, we assume that the force vector is in a plane parallel to the image plane in 3D space. Finally, we apply the SPH equation to smooth the forces across the particles

$$\bar{f}_t^{i,obs} = \sum_j m_j \frac{f_t^{j,obs}}{\rho_j} W(|r_i - r_j|, h).$$

The resulting force $\bar{f}_t^{i,obs}$ is then added to the other forces described in section 7.1 and S is computed as normal.

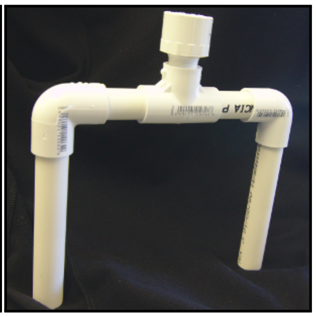
7.3 Experimental Setup

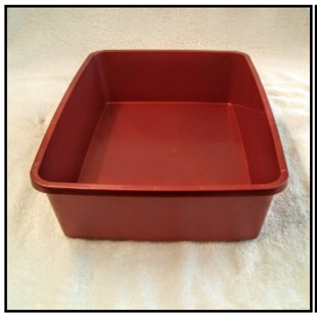
7.3.1 Robot & Sensors

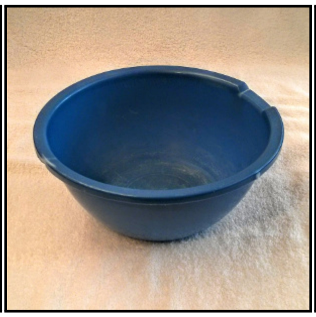
The robot used in the experiments in this chapter was an upper-torso robot with two 7-DOF arms, each with an electric parallel gripper. A table was fixed in front of the robot. To sense its environment, the robot used its Asus Xtion Pro RGBD camera, which recorded both color and depth images at 640×480 resolution at 30 Hz during each interaction, and its Infrared Cameras Inc. 8640P Thermal Imaging camera, which recorded thermographic

(a) *Cup*

(b) *Bottle*


(c) *Pipe Junction*


(d) *Pan*


(e) *Bowl*

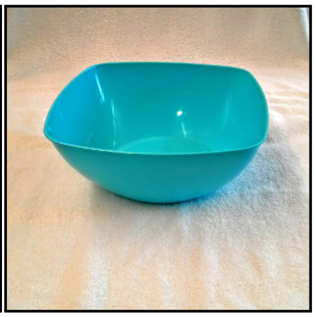

(f) *Fruit Bowl*

Figure 7.2: Objects used during the experiments. The top row shows the two containers the robot poured from as well as the pipe junction. The leftmost bowl in the bottom row was used in the pouring and the right two were used during the pipe junction experiments.

images at 640 × 512 resolution at 30 Hz during each interaction. The thermal camera was used in combination with heated water to acquire the ground truth pixel labelings. The cameras were locked in fixed relative positions and placed just below the robot's head at approximately chest height.

7.3.2 Data Collection

Pouring

We collected 16 pouring interactions. We varied the source container (*cup*, Figure 7.2a, or *bottle*, Figure 7.2b) and its initial fill amount (empty, 30%, 60%, or 90% full). Before each

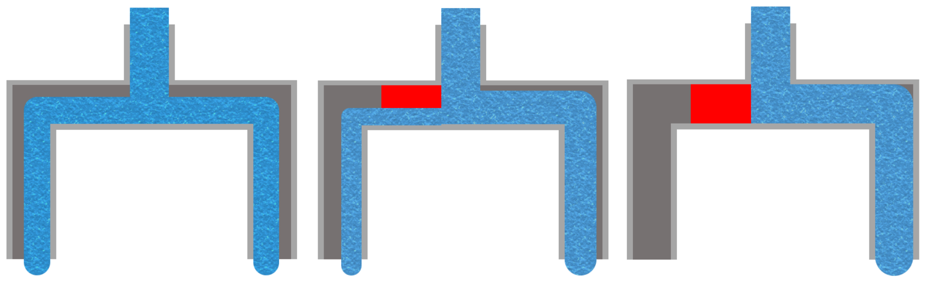


Figure 7.3: The 3 types of blockages placed in the pipe junction. (left to right) Pipe junction with no blockage; left leg is partially blocked; and left leg is fully blocked.

pouring interaction, a bowl (the *pan*, Figure 7.2d) was placed on the table in front of the robot. Next the source was placed in the robot's gripper, filled with water, and the gripper moved over the bowl. The robot then proceeded to rotate it's wrist along a fixed trajectory such that the opening of the container tilted down towards the bowl and water poured out. During each pouring interaction, the robot recorded from its RGBD and thermal cameras as well its joint poses. We collected two trials for each combination of source container and fill amount.

Pipe Junction

We collected 5 pipe junctions interactions. Before each of the pipe junction interactions, two bowls (*bowl*, Figure 7.2e, and *fruit bowl*, Figure 7.2f) were placed side-by-side on the table in front of the robot. Next, the robot held the ends of the pipe junction (Figure 7.2c) with its grippers over the bowls and recorded from its RGBD and thermal cameras while 1 liter of water was poured in the top opening. Each leg of the pipe junction could be fully blocked or partially blocked, i.e., the flow going to that leg could be partially restricted or entirely stopped. A diagram of the pipe junction and how the blockages affected flow is shown in Figure 7.3. The blockage can be placed in either leg, for a total of 5 possible configurations.

7.3.3 Data Processing

Before we can use our simulators to track the flow of liquid in the interactions described in the previous section, we must first perform some post-processing on the data. First, both the open-loop and closed-loop simulators require the object poses to be known over the course of the interaction. We utilize an object tracking method based on point cloud data to do this. Second, both closed-loop simulators require an image with pixels labels for the liquid. We use a thermal camera to acquire this labeling. We perform these steps offline, however both are capable of operating in real-time in online situations.

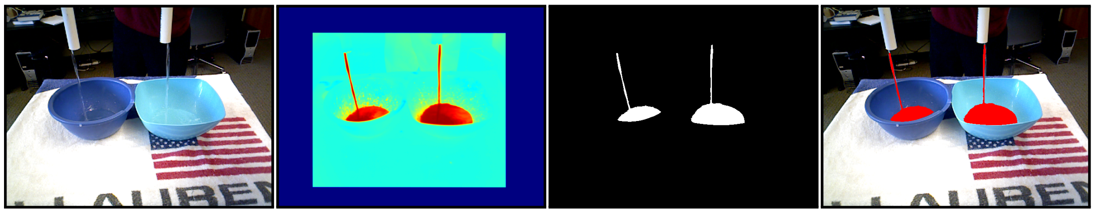
Object Tracking

We use the software program DART [136] (Dense Articulated Real-Time Tracking) to track the objects in each interaction. DART uses depth images to track objects over time. We initialize the pose of the bowls by using the Point Cloud Library's [128] built-in tabletop segmentation algorithm to find the point cluster on the table, and then set their initial pose to the centroid. We initialize the containers by computing the robot's forward kinematics to find the gripper pose. Once initialized, DART returns a pose for each object at each point in time over the interaction.

Liquid Labeling

For each pouring and pipe junction interaction, the water was heated to a temperature significantly above the surrounding environment but below its boiling point. The interactions were recorded with a thermal camera, and the thermal image was simply thresholded to locate the liquid pixels. Figure 7.4b shows an example thermal image recorded during a pipe junction interaction, and Figure 7.4c shows its corresponding thresholded values.

In addition to generating labels from the thermal image, it must also be calibrated to the depth image (the object poses generated by DART, and thus the entire simulator, operate in the depth camera frame of reference). That is, for each pixel in the thermal camera, we



(a) *RGB* (b) *Thermal* (c) *Threshold* (d) *Overlay*

Figure 7.4: Acquiring liquid labels from the thermal camera. The leftmost image is a color image of the scene, the center-left shows the corresponding thermal image transformed to the color image's space. The center-right image shows the liquid labels acquired via thresholding the thermal image, and the rightmost shows the labels overlayed on the color image.

must determine which pixel in the depth camera it corresponds to. This is not as simple as it may appear. Water is not visible in the depth image as the projected infrared light does not reflect properly off the surface. However, our depth camera also collects color images and calibrates it to the depth frame automatically. We can use the color image then to calibrate the thermal camera.

While there exist methods for doing a full registration between color and thermal images [116], these tend to be noisy and unreliable. In this chapter, because the water remains at a fixed distance from the camera, we use a simpler solution. First we take a checkerboard pattern printed on a wooden board and place it under a high-intensity halogen lamp. The light and dark pattern on the board absorbs light from the lamp at different rates, causing the dark squares to heat faster than the light squares. We then hold this board in front of both the thermal and color cameras at the same distance as the water. The differential heating causes the checkerboard pattern to be visible in both cameras, allowing us to find correspondence points between the two images. We then use these points to compute an affine transformation between the images, and use it to transform the thermal image onto the color image. Figures 7.4a and 7.4b show an example color image and its corresponding thermal image transformed onto the color space (the thermal camera has a narrower field of view than the color camera, which is why there are no thermal values around the edge of Figure 7.4b). Figure 7.4d shows the thresholded thermal image overlayed onto the color image.

7.3.4 Evaluation Criteria

We use two criteria for evaluating our methodology. The first is intersection over union (IOU). In this case, the state of the liquid simulation is projected into the camera by placing small spheres at each particle location and projecting those into the camera, taking into account occlusions by objects. We then compare the set of pixels labeled as *liquid* by this projection to the set of pixels labeled as *liquid* by the thermal image. The IOU is simply the intersection of these two sets divided by the union.

When comparing the probability of multiple simulations for the purposes of estimating hidden state, we use $P(\hat{\mathcal{I}}_\pi|\mathcal{I}_\pi)$ where $\hat{\mathcal{I}}_\pi$ is a set of predicted images for interaction π, and $\mathcal{I}_\pi$ is the set of ground truth images. To compute this, we first apply Bayes rule

$$P(\hat{\mathcal{I}}_\pi|\mathcal{I}_\pi) \propto P(\mathcal{I}_\pi|\hat{\mathcal{I}}_\pi)P(\hat{\mathcal{I}}_\pi).$$

For our experiments, we assume the prior $P(\hat{\mathcal{I}}_\pi)$ is uniform. To compute $P(\mathcal{I}_\pi|\hat{\mathcal{I}}_\pi)$, we assume each pixel is independent and simply multiply their individual probabilities together

$$P(\mathcal{I}_\pi|\hat{\mathcal{I}}_\pi) = \prod_{t=1}^{T}\prod_{j} P(j|\hat{j})$$

where we set $P(j|\hat{j})$ equal to δ if j and $\hat{j}$ are equal (both *liquid* or both *not-liquid*), and to $1-\delta$ if they are not. Due the the large number of pixels across all images and timesteps, we set $\delta = 0.50001$ to prevent underflow[1]. After computing the probabilities, we then normalize them so they sum to 1.

[1]Even in log-space, values would still periodically underflow with higher values for δ due to the large quantity of pixels.

	Open Loop	**MAP Filter**	**Warp Field**
Cup	60.17%	73.38%	75.94%
Bottle	67.25%	77.12%	79.41%
30%	35.56%	65.22%	67.01%
60%	77.62%	79.85%	82.80%
90%	77.94%	80.69%	83.22%
Overall	65.66%	76.03%	78.41%

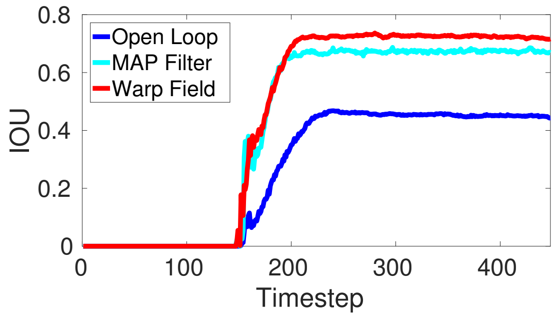


Figure 7.5: The table shows the IOU for each method. The graph shows the IOU at every timestep across one of the pouring experiments (*bottle* filled to *30%*).

7.4 Experiments & Results

We ran three experiments to evaluate our simulators at tracking the state of real-world liquids. The first utilized the pouring interactions and focused on quantitatively evaluating the open and closed loop simulators. The second and third experiments test our simulation methods at estimating the state of an unknown variable in the environment. This is an important ability for a robot, as often liquids are occluded by containers or other objects, forcing robots to reason about the hidden state of the liquids based on outcomes during an interaction, something that is not always necessary during rigid object interactions. Our second two experiments examine two different cases of hidden state estimation using liquids.

7.4.1 Comparing Open and Closed Loop Simulation Methods

To compare each of the three simulation methods (open loop, MAP filter, and warp field), we simulated them on the data collected for each pouring interaction. At the start of each interaction, we fill the 3D model of the container with the same amount of liquid as was filled in the real container. To do this, we perform binary search on the initial number of particles, running the simulation forward, holding the object poses constant, until each has settled and then computing the level of the liquid in the container. We then simulate the liquid forward in time, updating the object poses based on the tracked poses acquired using DART. We evaluate each method by comparing their IOUs, computed as described in section 7.3.4[2].

The IOU for the three simulation methods is shown in the table in Figure 7.5. The upper two rows show the IOU for the methods conditioned on the two types of containers used; The middle rows show the IOU conditioned on the initial percent full of the container; and the last row shows the overall IOU for each method. This table reveals some interesting phenomena. It is not immediately clear why all the simulators seem to perform slightly better on interactions where the robot poured from the bottle rather than the cup. However, the middle of the table shows that all of the methods tend to perform better when more liquid is involved. We notice that the *bottle*, while having a similar diameter as the *cup*, is taller, meaning if they are filled to the same ratio full (e.g., 30%), then the *bottle* will have more overall liquid. This explains the slight bump in performance from one container to the other.

The most important revelation, however, is that both closed-loop simulation methods outperform the open-loop simulation by a significant margin. This is illustrated graphically by the graph on the right in Figure 7.5, which shows the IOU at every timestep over one sequence, and clearly shows that the closed-loop methods are better able to match the location of the real liquid than the open-loop method. Additionally, both the table and the graph show that the warp field method outperforms the MAP filter method. This clearly shows that closing the loop in liquid simulations can make the trajectory of the liquid better match real world liquid dynamics.

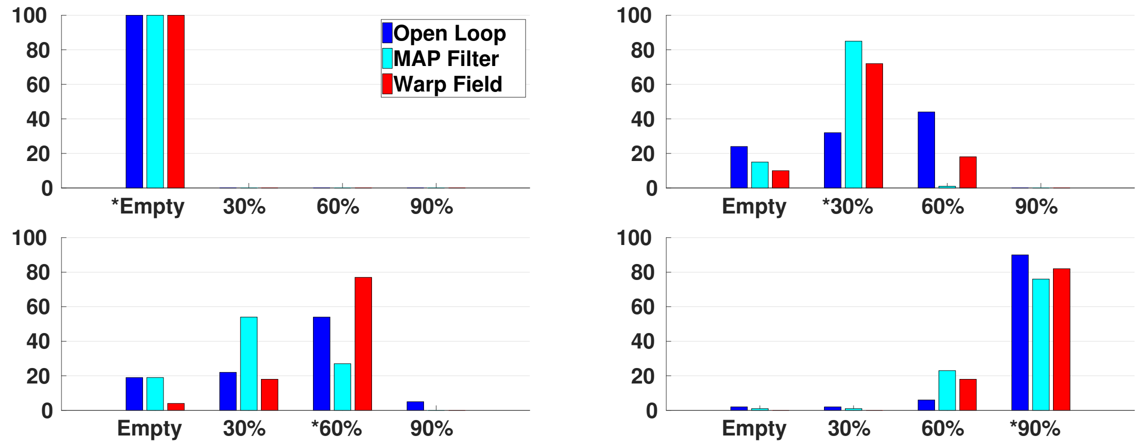


Figure 7.6: Probability distribution over the estimated initial fill amounts. They are aggregated by the true fill amounts. From left to right and top to bottom they are empty, 30% full, 60% full, and 90% full (indicated by the *). The blue bars show results from the open loop method, cyan for the MAP filter, and red for the warp field.

7.4.2 Estimating the Initial Amount of Liquid

We evaluated all three simulation methods on the same hidden state task. For each pouring interaction, the initial amount of liquid in the container was not given to the robot. Instead, the task of the robot was to estimate this amount based on the observations and its own liquid simulations. To do this, the robot needs to run multiple simulations for each interaction, one for each possible fill amount, and compare the predictions of each simulation to the observation.

For each pouring interaction, the robot ran 4 simulations: one where the container was left empty, one where the container was filled to 30% full, one where the container was filled to 60% full, and one where it was filled to 90% full. For each simulation, the liquid particles are simulated forward in time as the object poses are updated via their tracked poses. We compute the probability of each simulation by evaluating the probability of their predicted

[2]The 4 pouring interactions where the container was left empty were not included in this analysis because the union part of the IOU would be 0, resulting in a division by 0.

images as described in section 7.3.4.

Figure 7.6 shows the results of performing this for each of the pouring interactions, aggregated by the ground truth fill amount (indicated by the * in the x-axis of each graph). The blue bars show the probability distributions for the open-loop method, the cyan bars show the distribution for the MAP filter method, and the red bars show the distribution for the warp field method. All methods are easily able to correctly place the highest probability on the empty simulation when there is in fact no liquid in the interaction, which follows intuition as there are no observed liquid particles. Additionally, even though there is slightly more confusion, all of the methods place the highest probability on the 90% simulation when the containers start out 90% full. Again, this aligns with intuition as it is easy to distinguish "a lot" of liquid from "almost no" liquid. The most confusion occurs when trying to distinguish "a little" (30%) from "some" (60%). The open loop method is almost completely unable to distinguish between the two, both distributions being very similar. The MAP filter method is slightly better, but still gets confused when the true amount in the container is 60%. Only the warp field method is able to correctly estimate the initial amount of liquid, placing over 70% probability on the correct simulation in every case.

7.4.3 Solving the Pipe Junction Task

The final experiment we performed was the pipe junction task. Here the task is for the robot to find the blockage in a pair of connected pipes simply by observing the liquid as it exits the pipes, a situation the robot may find itself in if, say, trying to diagnose a broken sink. We assume that the robot knows *a priori* the default, unblocked flow rate of liquid through the pipes, and thus must use the change in flow to find the blockage. To test this, a pipe T-junction was held inverted over two bowls such that the legs of the T emptied into different bowls, both visible to the robot. However, the task is to find the blockage based only on the output of the pipes, so the T-junction was held high enough so that the robot could only see the openings on the bottom and not the top opening. To simulate a constant flow into the pipes, a container with exactly 1 liter of water was tilted at a constant angular velocity so that the liquid flowed into the top opening of the junction. The type of blockage used (if any), unblocked, partially blocked, or blocked, was placed inside the pipe, not visible to the robot. We used the data collected during the pipe junction interactions to evaluate the robot on this task.

To solve this task, like in the previous experiment, the robot needs to run multiple simulations with different values for the hidden state (the pipe blockages) and compare their outcomes. For each interaction, the robot ran 5 simulations: one for both legs unblocked, one for the right leg partially blocked, one for the right leg fully blocked, one for the left leg partially blocked, and one for the left leg fully blocked. The probability of each simulation is computed using the method described in section 7.3.4.

Figure 7.7 shows the probability for each of the simulated blockages over time for one of the interactions using the best closed-loop method (warp field). The robot ran one simulation for each blockage type, and the diagrams across the top of the figure indicate where the blockage in that simulation was placed. The color bordering each diagram corresponds to the color of the line indicating that simulation's probability over time. After only a short time window, the robot is able to place 100% probability on the correct blockage (*partial-left*). Indeed, we ran this on all 5 pipe junction interactions, and by the end of each, the robot had placed 100% on the correct blockage in every case. We also evaluated the 5 interactions using the open-loop method. It was able to correctly estimate with 100% probability in the simpler cases (no blockage or fully blocked) as would be expected. However, for the more difficult interactions (partial blockage), it only picked the correct blockage type and location in one case (when the true blockage was *partial-left*) and in the other case incorrectly placed 100% probability on there being no blockage. While the point of this experiment was to show the possible type of reasoning that can be done with full physics-based liquid models, even here the closed-loop methods outperform the open-loop methods, if only in 1 out of 5 cases. Regardless, by using the closed-loop liquid simulation methods developed here, the robot is clearly able to robustly solve this task.

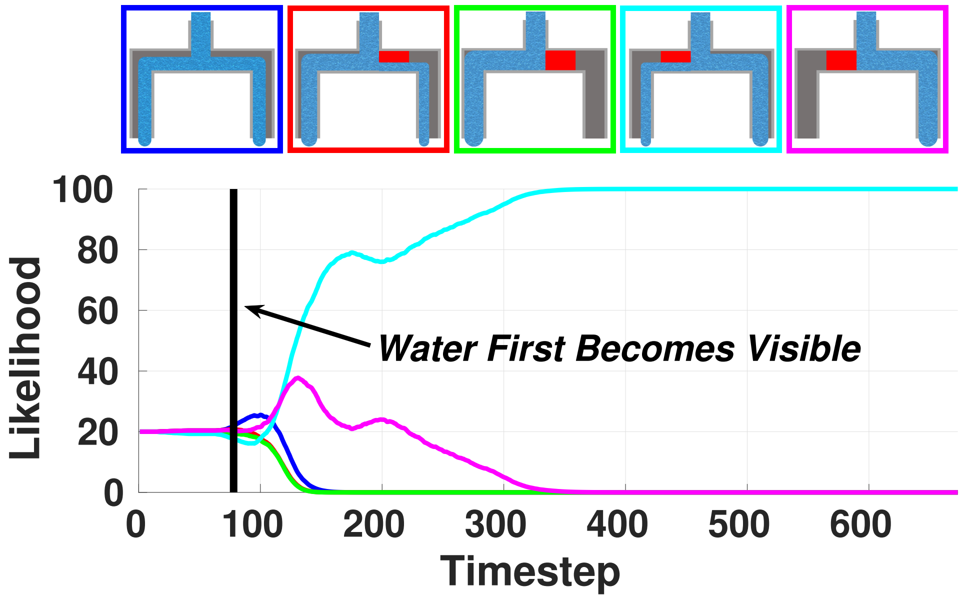


Figure 7.7: Probability distribution over the blockage location over time for a single interaction. The 5 diagrams across the top correspond to the five different simulations the robot ran, each color-coded to the corresponding line in the plot. The true blockage was placed in the left leg and only partially blocked the leg (in the keys in the top row, second from right). Best viewed in color.

7.5 Discussion

In this chapter, we proposed two methods for tracking the state of liquid with a closed-loop simulator. The first, inspired by Bayes filter techniques in robotics, used a MAP filter to correct errors in the simulator. The second, inspired by the physical forces underlying the simulator, applied a warp field to the particles to correct the error. The results clearly show that both our closed-loop methods are better at tracking the liquid than the open-loop method. We also showed how these closed-loop simulations can be used to reason about and infer the hidden variables of an interaction involving liquids. To our knowledge, this is the first time real liquid observations have been combined with liquid simulations for robotics tasks.

From the results in this chapter, there are several important insights to take away:

Reasoning about Liquids: So far, reasoning about liquids applied to real robots has been limited to restricted tasks such as pouring in both prior work [127, 18] and in our previous chapters. With our physics-based model, reasoning about liquids can be done on a much wider variety of tasks. The last two experiments in this chapter both involve completely different tasks, one reasoning about pouring, the other about blockages in pipes, yet the same algorithm is able to solve both tasks, without any special knowledge aside from generic 3D models. This is the kind of reasoning that is only enabled through full, physics-based models. Another advantage of our method over our previous deep learning methods or even a non-physics model-based approach [163] is that the persistence of a liquid is trivially inferred. For example, a robot using this model could observe a pouring interaction, and it would be immediately obvious that the new liquid in the target container originated in the source container, and that the overall liquid is the same at the end of the pour as it was at the beginning.

Generalizing to Other Liquids: Another advantage of a physics-based model is that it can generalize to different types of liquid. Yamaguchi and Atkeson [163] developed a model-based detector that could determine the location of liquids in a scene, and they showed that it could generalize to a wide array of liquid types. This is unlike learning-based models, which cannot generalize to liquids too different from their training set. With the alteration of a few physical parameters, a physics-based model can generalize to liquids as diverse as water, oil, honey, and even dough. It is currently an open challenge as to how to infer these parameters efficiently from observation. We investigate this in a simulated setting in the next chapter.

Predicting Liquid Behavior: While others have used physics-based models for liquids [74], none have yet combined them with real perception. As a result, due to the quick divergence of open-loop models with reality, there has been little prior work exploring the possible action spaces around liquids. Closed-loop liquid simulations enable robots to use the same model to interact with liquids in a wide variety of settings, such as carrying a container across a room without spilling its contained liquid, scooping liquid with a spoon, and ejecting liquid from a syringe in a controlled manner. Without closed-loop liquid simulations, each of these tasks would require developing a separate model. Using an algorithm such as model predictive control [19], the robot could plan for a short time horizon into the future using the open-loop simulation, but track the current state using the closed-loop simulation, thus preventing a fatal divergence from reality.

Overall, this chapter showed the power of physics-based models for 3D reasoning about liquids. In previous chapters in this thesis, we showed how deep learning methods can be used to interact with liquids. Our results up to this point in the thesis show that both methods can be very useful for robots. In the next chapter, we focus on a method to combine them into a single model.

7.5.1 Practical Takeaways

Implementing a liquid simulator to track real liquid poses several technical challenges not often found in a pure simulation environment. One of those is the issue of particle clipping. Particle clipping is when, at the start of a timestep, a particle is on one side of an object mesh, and then after the update in the timestep is on the other. In this scenario, even though the particle's straight line path would put it in collision with the mesh, because it moves quickly, at no point does the simulator detect a collision. The result of this is particles that appear to move through meshes with thin walls, which are many of the objects used in this chapter such as cups and bottles. This is of course not desirable, and so we employed two primary strategies to combat it. First, we computed ray-plane intersections between the movement vector of the particle and the triangles in the 3D mesh to catch potential collisions. While in theory this should fix all issues relating to this problem, in practice with highly triangulated meshes doing so can be extraordinarily slow. We used an approximate solution to compute intersections for only nearby triangles, not all triangles, which caught most issues. To ensure that no particle ever clipped through a mesh, we also broke any large movements into smaller movements so we could check at each point along the vector. In pure simulation environments, objects with thin walls are often not used for this explicit reason. However here we could not get around the issue because the real objects the robot used were containers with relatively thin walls. But this solution we developed was effective at fixing the issue, although it did significantly decrease the speed of the simulation, which is a large part why our simulations in this chapter do not necessarily all run at real-time.

Another challenge we encountered while developing the methodology in this chapter was getting the particles to cover the entire perceived liquid. When developing the closed-loop methodology, the robot has to deal with two types of mismatch between the liquid particles and the perceived liquid pixels: 1) instances where a particle has no corresponding liquid pixel, and 2) instances where a liquid pixel has no corresponding particle. For a method like our warp field method, applying a force for each liquid pixel to attract particles to them works well enough to fix mismatch due to 1. However, this often results in particles congregating around a few liquid pixels, leaving plenty of other liquid pixels without any corresponding particles. To solve this, we introduced the term Φ to measure how well "covered" any liquid pixel was. This could then be used to encourage particles to better solve mismatch due to 2. In practice, by balancing the contribution of Φ to each of the closed loop methods, we were able to encourage particles to spread out to cover the liquid pixels without creating undesirable oscillations.

Chapter 8

SPNETS: COMBINING DEEP LEARNING WITH LIQUID SIMULATION

In previous chapters in this thesis, we have taken two distinct approaches to robotic interaction with liquids. In chapters 3 to 6 we examined learning-based approaches using deep neural networks to enable robots to perceive, reason about, and manipulate liquids. In the previous chapter, chapter 7, we looked at how physics-based models can be used to perform reasoning tasks with liquids. In all cases, our results showed that these methods can be used by robots to solve tasks involving liquids. However, up to this point, they have been largely separate.

In this chapter we propose to combine the structure of analytical fluid dynamics models with the tools of deep neural networks to enable robots to interact with liquids. Specifically, we propose Smooth Particle Networks (SPNets), which adds two new layers, the ConvSP layer and the ConvSDF layer, to the deep learning toolbox. These layers allow networks to interface directly with unordered sets of particles (our chosen representation for liquids). We then show how we can use these two new layers, along with standard layers, to directly implement fluid dynamics using Position Based Fluids (PBF) [90] inside the network, where the parameters of the network are the fluid parameters themselves (e.g., viscosity or cohesion). Because we implement fluid dynamics as a neural network, this allows us to compute full analytical gradients. We evaluate our fully differentiable fluid model in the form of a deep neural network on the tasks of learning fluid parameters from data, manipulating liquids, and learning a policy to manipulate liquids. In this chapter, the final experimental chapter of this thesis, we make the following contributions 1) a fluid dynamics model that can interface directly with neural networks and is fully differentiable, 2) a method for learning fluid

```
1:  function UpdateFluid(P, V)
2:      $V' = \text{ApplyForces}(V)$
3:      $P' = P + \frac{V'}{\Delta t}$
4:      while $\neg\text{ConstraintsSatisfied}(P')$ do
5:          $\Delta P^{\omega} = \text{SolvePressure}(P')$
6:          $\Delta P^{c} = \text{SolveCohesion}(P')$
7:          $\Delta P^{s} = \text{SolveSurfaceTension}(P')$
8:          $P' = P' + \Delta P^{\omega} + \Delta P^{c} + \Delta P^{s}$
9:          $P' = \text{SolveObjectCollisions}(P')$
10:     end while
11:     $V' = \frac{P'-P}{\Delta T}$
12:     $V' = V' + \text{ApplyViscosity}(P', V')$
13:     return $P', V'$
14: end function
```

Figure 8.1: The PBF algorithm. P is the list of particle locations, V is the list of particle velocities, and ΔT is the timestep duration.

parameters from data using this model, and 3) a method for using this model to manipulate liquid by specifying its target state rather than through auxiliary functions.

The rest of this chapter is laid out as follows. The next section describes Position Based Fluids (PBF), the fluid dynamics algorithm we implement here. The section after that describes Smooth Particle Networks (SPNets), our implementation of PBF using the deep learning toolbox. Next we describe how we evaluate SPNets and the results we get, and finally we end the chapter with a discussion of the insights we gained.

8.1 Position Based Fluids

In this chapter, we implement fluid dynamics using Position Based Fluids (PBF) [90]. PBF is a Lagrangian approximation of the Navier-Stokes equations for incompressible fluids [2]. That is, PBF uses a large collection of particles to represent incompressible fluids such as water, where each particle can be thought of as an individual "packet" of fluid. We chose a particle-based representation for our fluids rather than a grid-based (Eulerian) representation as for sparse fluids, particles have better resolution for fewer computational resources. We briefly describe PBF here and direct the reader to [90] for details.

The key difference between PBF and Smoothed Particle Hydrodynamics (SPH) [37], which is often used to do particle-based fluid simulations (and which we used in the previous chapter), is that SPH only applies to compressible fluids. This is because SPH models properties of the fluid, such as pressure, as forces (e.g., places of high pressure apply a force pushing particles away), which necessarily means that fluids modeled with SPH can compress so long as the compressing force exceeds the fluid's internal forces. PBF instead treats these properties as constraints (e.g., the pressure must remain constant throughout the fluid), and then iteratively attempts to compute the "correction" ΔP to each particle's position that satisfies these constraints. In this way, PBF is designed to be used for incompressible fluids such as water, which is why we chose to use it in this chapter.

Figure 8.1 shows a general outline of the PBF algorithm. At each timestep, the update for each particle is computed as follows. First external forces are applied to the particles by adding their accelerations to the velocities (line 2). For this paper, the only external force is gravity. Then, on line 3, the particles are moved forward according to their velocities. Next the inner loop (lines 5–9) solves the incompressibility constraints (pressure, cohesion, and surface tension) and then fixes collisions with static objects. The constraints are solved iteratively using a loop rather than in one-shot to facilitate convergence. After the loop, the velocity of the particles is updated to reflect their actual change in position over time given the constraints (line 11). Finally, the viscosity force is computed and applied to the velocities of the particles (line 12). The result of all this is the new particle positions P' and velocities V' after updating their dynamics for one timestep.

Figure 8.1 gives a high-level overview of the PBF algorithm. It relies on several external functions, which we will describe in the following paragraphs. For the purposes of this paper, the function ConstraintsSatisfied returns true after a fixed number of iterations, which here we set to be 3. Furthermore, for the function SolveObjectCollisions, we determine if the vector of movement for each particle collides with the mesh of an object, and if so we simply adjust the vector so that the particle stops at the object boundary. The remaining functions, SolvePressure, SolveCohesion, SolveSurfaceTension, and ApplyViscosity, are used to apply the fundamental physical properties to the particles such as pressure or surface tension.

The pressure correction δp_i^ω for each particle i is computed by the function SolvePressure to satisfy the constant pressure constraint. Intuitively, the pressure correction step finds particles with pressure higher than the constraint allows (i.e., particles where the density is greater than the ambient density), then moves them along a vector away from other high pressure particles, thus reducing the overall pressure and satisfying the constraint. The pressure correction δp_i^ω for each particle i is computed as follows

$$\delta p_i^\omega = \sum_{j \in P - \{i\}} n_{ji}(\omega_i + \omega_j) W_\omega(d_{ij}, h) \tag{8.1}$$

where n_{ji} is the normalized vector from particle j to particle i, ω_k is the pressure at particle k, W_ω is a kernel function, d_{ij} is the distance from i to j, and h is the cutoff for W_ω (that is, for all particles further than h apart, W_ω is 0). The pressure ω_k at every particle k is computed as

$$\omega_k = \lambda_\omega \max(\rho_k - \rho_0, 0) \tag{8.2}$$

where λ_ω is the pressure constant, ρ_k is the density of the fluid at particle k, and ρ_0 is the rest density of the fluid. The density ρ_k is computed the same in PBF and SPH as follows

$$\rho_k = \sum_{j \in P} m_j W_\rho(d_{kj}, h) \tag{8.3}$$

where m_j is the mass of particle j and W_ρ is a kernel function. For W_ω we use $\frac{30}{\pi h^3}\left(1-\frac{d}{h}\right)\frac{1}{h}$ and for W_ρ we use $\frac{15}{\pi h^3}\left(1-\frac{d}{h}\right)^2$, the same as used in [90].

The cohesion correction δp_i^c for each particle i as computed by SolveCohesion is

$$\delta p_i^c = \sum_{j\in P-\{i\}} \lambda_c n_{ij} W_c(d_{ij}, h)$$

where λ_c is the cohesion constant and W_c is a kernel function. For W_c we use the same kernel as [90]

$$W_c(d,h) = \frac{-(1-d_0)}{d_0^2}\left(\frac{d}{h}\right)^3 + \frac{d_0^2+d_0+1}{d_0^2}\left(\frac{d}{h}\right)^2 - 1$$

where d_0 is the fluid rest distance as a fraction of h. For this paper we fix d_0 to 0.5.

The surface tension correction δp_i^s for each particle i as computed by SolveSurfaceTension is computed using the following equation

$$\delta p_i^c = \sum_{j\in P-\{i\}} \frac{\lambda_s}{\rho_0}(n_j - n_i) I(d_{ij} \leq h)$$

where λ_s is the surface tension constant, n_k is the normal of the fluid surface at particle k, and I is the indicator function. The normal n_k is computed as

$$n_k = \sum_{j\in P-\{i\}} n_{ij} W_c(d_{ij}, h)$$

where W_c is the same kernel function used for the cohesion constraint.

Finally, the viscosity update δv_i for each particle i computed by ApplyViscosity is

$$\delta v_i = \sum_{j\in P-\{i\}} \frac{\lambda_v}{\rho_0}(v_j - v_i) W_\rho(d_{ij}, h)$$

where λ_v is the viscosity constant, v_k is the velocity of particle k, and W_ρ is the same kernel function used to compute the density.

To compute the next set of particle locations P' and velocities V' from the current set P, V, these functions are applied as described in the equation in figure 8.1. For the experiments in this paper, the constants are empirically determined and we set h to 0.1.

8.2 *Smooth Particle Networks*

In this chapter, we wish to implement Position Based Fluids (PBF) using deep learning tools. However current standard neural networks lack the functionality to implement PBF inside the network structure. While [118] developed PointNet layers which work on unordered point sets, these were designed for object recognition and segmentation and are not well suited for fluid dynamics. Thus, in order to implement PBF in a network, we need to add new functionality. Specifically, we propose two new neural network layers for handling unordered sets of particles. The first is the ConvSP layer, which computes particle-particle pairwise interactions, and the second is the ConvSDF layer, which computes particle-static object interactions[1]. We combine these two layers with standard operators (e.g., elementwise addition) to reproduce the algorithm in figure 8.1 inside a deep network. Since we exactly reproduce PBF inside the network, the parameters are the λ_* constants descried in section 8.1. We implemented both our layers with support for GPUs to enable efficient processing times and with native support for analytical gradients. We used PyTorch [113] to construct the fluid dynamics by combining our layers with standard network layers, which also enables us to take advantage of its automatic differentiation for our PBF implementation.

8.2.1 ConvSP

The ConvSP layer is designed to compute particle to particle interactions. To do this, we implement the layer as a smoothing kernel over the set of particles. That is, ConvSP computes the following

$$ConvSP(X, Y) = \left\{ \sum_{j \in X} y_j W(d_{ij}, h) \;\middle|\; i \in X \right\}$$

where X is the set of particle locations and Y is a corresponding set of feature vectors[2], y_j is the feature vector in Y associated with j, W is a kernel function, d_{ij} is the distance between

[1]The code for SPNets is available at `https://github.com/cschenck/SmoothParticleNets`

[2]In general, these features can represent any arbitrary value, however for the purposes of this paper, we use them to represent physical properties of the particles, e.g., mass or density.

particles i and j, and h is the cutoff radius (i.e., for all $d_{ij} > h$, $W(d_{ij}, h) = 0$). This function computes the smoothed values over Y for each particle using W.

While this function is relatively simple, it is enough to enable the network to compute the solutions for pressure, cohesion, surface tension, and viscosity (lines 5–7 and 12 in figure 8.1). In the following paragraphs we will describe how to compute the pressure solution using the ConvSP layer. Computing the other 3 solutions is nearly identical.

To compute the pressure correction solution in equation (8.1) above, we must first compute the density ρ_k at each particle k. Equation (8.3) describes how to compute the density. This equation closely matches the ConvSP equation from above. To compute the density at each particle, we can simply call $ConvSP(P, M)$, where P is the set of particle locations and M is the corresponding set of particle masses. Next, to compute the pressure ω_k at each particle k as described in equation (8.2), we can use an elementwise subtraction to compute $\rho_k - \rho_0$, a rectified linear unit to compute the max, and finally an elementwise multiplication to multiply by λ_ω. This results in Ω, the set containing the pressure for every particle.

Plugging these values into equation (8.1) is not as straightforward. It is not obvious how the term $n_{ji}(\omega_i + \omega_j)$ could be represented by Y from the ConvSP equation. However, by unfolding the terms and distributing the sum we can represent equation (8.1) using ConvSP.

First, note that the vector n_{ji} is simply the difference in position between particles i and j divided by their distance. Thus we can replace n_{ji} as follows

$$\delta p_i^\omega = \sum_{j \in P - \{i\}} \frac{p_i - p_j}{d_{ij}} (\omega_i + \omega_j) W_\omega(d_{ij}, h)$$

where p_k is the location of particle k. For simplicity, let us incorporate the denominator d_{ij} into W_ω to get it out of the way. We define $\overline{W}_\omega(d_{ij}, h) = \frac{1}{d_{ij}} W_\omega(d_{ij}, h)$. This results in

$$\delta p_i^\omega = \sum_{j \in P - \{i\}} (p_i - p_j)(\omega_i + \omega_j) \overline{W}_\omega(d_{ij}, h).$$

Next we distribute the terms in the parentheses to get

$$\delta p_i^\omega = \sum_{j \in P - \{i\}} (p_i \omega_i + p_i \omega_j - p_j \omega_i - p_j \omega_j) \overline{W}_\omega(d_{ij}, h).$$

We can now rearrange the summation and distribute $\overline{W}_\omega$ to yield

$$\delta p_i^\omega = p_i \omega_i \sum \overline{W}_\omega(d_{ij}, h) + p_i \sum \omega_j \overline{W}_\omega(d_{ij}, h) - \omega_i \sum p_j \overline{W}_\omega(d_{ij}, h) - \sum p_j \omega_j \overline{W}_\omega(d_{ij}, h).$$

Here we omitted the summation term $j \in P - \{i\}$ from our notation for clarity. We can compute this over all i using the ConvSP layer as follows

$$\Delta P^\omega = P * \Omega * ConvSP(P, \{1\}) + P * ConvSP(P, \Omega) - \Omega * ConvSP(P, P) - ConvSP(P, P * \Omega)$$

where $*$ represents elementwise multiplication and $+$ and $-$ are elementwise addition and subtraction respectively. $\{1\}$ is a set containing all 1s.

8.2.2 ConvSDF

The second layer we add is the ConvSDF layer. This layer is designed specifically to compute interactions between the particles and static objects in the scene (line 9 in figure 8.1). We represent these static objects using signed distance functions (SDFs). The value $SDF(p)$, where p is a point in space, is defined as the distance from p to the closest point on the object's surface. If p is inside the object, then $SDF(p)$ is negative.

We define K to be the set of offsets for a given convolutional kernel. For example, for a 1×3 kernel in 2D, $K = \{(0, -1), (0, 0), (0, 1)\}$. ConvSDF is defined as

$$ConvSDF(X) = \left\{ \sum_{k \in K} w_k \min_j SDF_j(p_i + k * d) \;\middle|\; i \in X \right\}$$

where w_k is the weight associated with kernel cell k, p_i is the location of particle i, SDF_j is the jth SDF in the scene (one per rigid object), and d is the dilation of the kernel (i.e., how far apart the kernel cells are from each other). Intuitively, ConvSDF places a convolutional kernel around each particle, evaluates the SDFs at each kernel cell, and then convolves those values with the kernel. The result is a single value for each particle.

We can use ConvSDF to solve object collisions as follows. First, we construct $ConvSDF_R$ which uses a size 1 kernel (that is, a convolutional kernel with exactly 1 cell). We set the weight for the single cell in that kernel to be 1. With a size 1 kernel and a weight w_k of 1, the summation, the kernel weight w_k, and the term $k * d$ fall out of the ConvSDF equation (above). The result is the SDF value at each particle location, i.e., the distance to the closest surface, where negative values indicate particles that have penetrated inside an object. We can compute that penetration R of the particles inside objects as

$$R = ReLU(-ConvSDF_R(P))$$

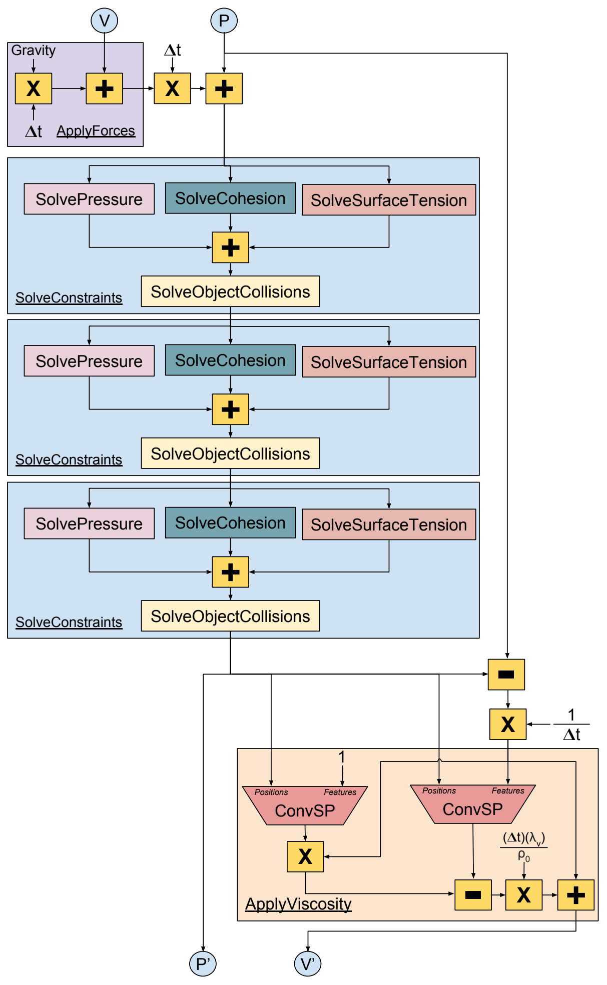
where $ReLU$ is a rectified linear unit. R now contains the minimum distance each particle would need to move to be placed outside an object, or 0 if the particle is already outside the object. Next, to determine which direction to "push" penetrating particles so they no longer penetrate, we need to find the direction to the surface of the object. Without loss of generality, we describe how to do this in 3D, but this method is applicable to any dimensionality. We construct $ConvSDF_X$, which uses a 3×1×1 kernel, i.e., 3 kernel cells all placed in a line along the X-axis. We set the kernel cell weights w_k to -1 for the cell towards the negative X-axis, +1 for the cell towards the positive X-axis, and 0 for the center cell. We construct $ConvSDF_Y$ and $ConvSDF_Z$ similarly for the Y and Z axes. By convolving each of these 3 layers, we use local differencing in each of the X, Y, and Z dimensions to compute the normal of the surface of the object n_{SDF}, i.e., the direction to "push" the particle in. We can then update the particle positions P' as follows

$$P' = P' + R * n_{SDF}.$$

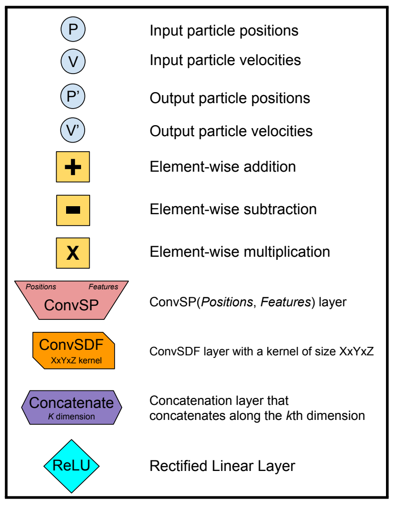
That is, we multiply the distance each particle is penetrating an object (R) by the direction to move it in (n_{SDF}) and add that to the particle positions.

8.2.3 Smooth Particle Networks (SPNets)

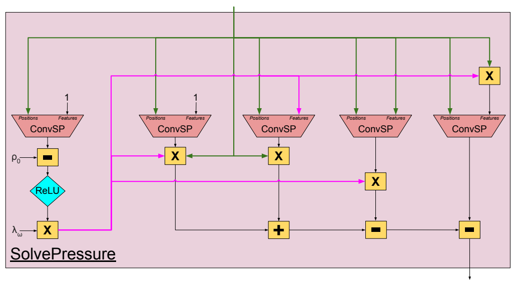
Using the ConvSP and ConvSDF layers described in the previous sections and standard network layers, we design SPNets to exactly replicate the PBF algorithm in figure 8.1. That is, at each timestep, the network takes in the current particle positions P and velocities V and computes the fluid dynamics by applying the algorithm line-by-line, resulting in new positions P' and velocities V'. We show an SPNet layout diagram figure 8.2.



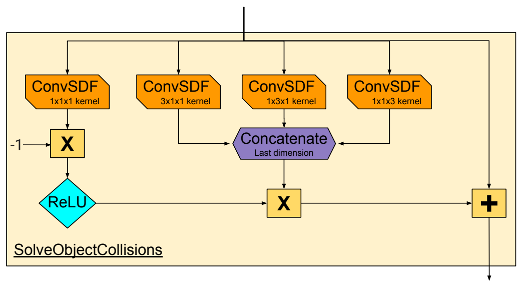
(a) SPNet


P Input particle positions
V Input particle velocities
P' Output particle positions
V' Output particle velocities
Element-wise addition
Element-wise subtraction
Element-wise multiplication
ConvSP(Positions, Features) layer
ConvSDF XxYxZ kernel: ConvSDF layer with a kernel of size XxYxZ
Concatenate K dimension: Concatenation layer that concatenates along the kth dimension
ReLU: Rectified Linear Layer

(b) Legend



(c) SolvePressure



(d) SolveObjectCollisions

Figure 8.2: The layout for SPNet. The upper-left shows the overall layout of the network. The functions SolvePressure, SolveCohesion, SolveSurfaceTension, and SolveObjectCollisions are collapsed for readability. The lower-right shows the expansion of the SolveObjectCollisions function, with the line in the top of the box being the input to the SolveObjectCollisions in the upper-left diagram and the line out of the bottom of the box being the line out of the box. The lower-left shows the expansion of the SolvePressure function. For clarity, the input line (which represents the particle positions) is colored green and the line representing the particle pressures is colored pink.

The first operation the network performs is to apply the external forces to the particles, line 2 of the PBF algorithm (shown in figure 8.1) and the lavender box in the upper-left of figure 8.2a. Next the network updates the particle positions according to their velocities, line 3 of PBF and the element-wise multiplication and addition immediately to the right of ApplyForces in the diagram. After this, the network iteratively solves the fluid constraints (lines 5–9), shown by the SolveConstraints boxes in figure 8.2a. Here we show 3 constraint solve iterations, however in principle the network could have any number. Each constraint solve partially updates the particle positions to better satisfy the given constraints.

We consider 3 constraints in this paper: pressure (line 5), cohesion (line 6), and surface tension (line 7). Each is shown as an individual box in figure 8.2a. Figure 8.2c shows the full network layout for the pressure constraint. This exactly computes the solutions to equations 8.1–8.3 from the previous section on PBF. Note the column under the leftmost ConvSP layer in figure 8.2a; it computes the pressure set Ω. This is then used to compute the result of the other 4 ConvSP layers. The final step of each constraint solve iteration is to solve the object collisions. The expansion of this box is shown in figure 8.2d. The ConvSDF layer on the left computes the particle penetration R into the SDFs, and the 3 on the right compute the normal n_{SDF} of the SDFs. Note that in this diagram we show the layout for particles in 3D (there are 3 ConvSDF layers on the right of figure 8.2d, one to compute the normal direction in each dimension), however this can applied to particles in any dimensionality.

After finishing the constraint solve iterations, the network computes the adjusted particle velocities based on how the positions were adjusted (line 11 of the PBF algorithm), shown in figure 8.2a as the element-wise subtraction and multiplication above the ApplyViscosity box. Finally, the network computes the viscosity, shown in the tan box in the bottom-right of figure 8.2a. Viscosity only affects the particles velocities, so the output positions of the particles are the same as computed by the constraint solver.

There are several parameters and constants in this network. In the ApplyForces box in the upper-left of figure 8.2a, *Gravity* is set to be $-9.8\frac{m}{s^2}$ and Δt is set to be $\frac{1}{60}$. The rest density ρ_0, shown in the ApplyViscosity box in the lower-right of figure 8.2a and in the

SolvePressure box in figure 8.2c, is set empirically based on the rest density of water. The fluid parameters λ_ω and λ_v are shown in figure 8.2c and the lower-right of figure 8.2a respectively.

By repeatedly applying the network to the new positions and velocities at each timestep, we can simulate the flow of liquid over time. We utilize elementwise layers, rectified linear layers (ReLU), and our two particle layers ConvSP and ConvSDF to compute each line in figure 8.1. Since elementwise and ReLU layers are differentiable, and because we implement analytical gradients for ConvSP and ConvSDF, we can use backpropagation through the whole network to compute the gradients. Additionally, our layers are implemented with graphics processor support, which means that a forward pass through our network takes approximately $\frac{1}{15}$ of a second for about 9,000 particles running on an Nvidia Titan Xp graphics card.

8.3 Evaluation & Results

To demonstrate the utility of SPNets, we evaluated it on four types of tasks, described in the following sections. First, we verify the correctness of our implementation of fluid dynamics by comparing it to a commercial fluid simulator. Then, we show how our model can learn fluid parameters from data. Next, we show how we can use the analytical gradients of our model to do liquid control. Finally, we show how we can use SPNets to train a reinforcement learning policy to solve a liquid manipulation task using policy gradients.

8.3.1 Model Comparison

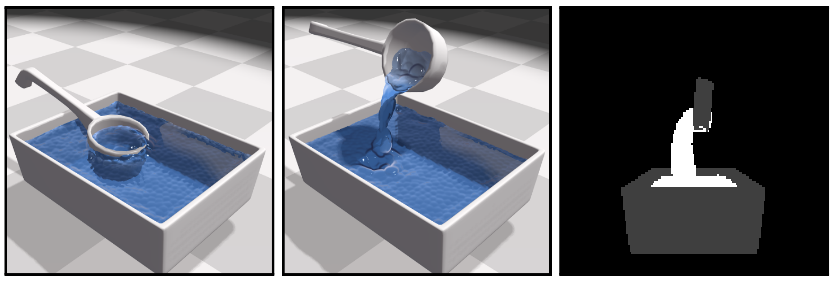
We compare our model to Nvidia FleX [91], a commercially available physics simulation engine which implements fluid dynamics using Position Based Fluids (PBF). For this comparison, we set all the model parameters (e.g., the pressure parameter λ_ω) to be the same for both FleX and SPNets, we initialize the particle poses and velocities to the same values, and all rigid objects follow the same trajectory. We compare FleX and SPNets on two scenes. In the *scooping* scene, the liquid rests at the bottom of a large basin as a cup moves in a circle, scooping liquid and then dumping it into the air. In the *ladle*, the liquid rests in a rectangular container as a ladle scoops some from the container and then pours it back into it. Images from the *ladle* scene are shown in figure 8.3a. We simulate each scene 9 times, once for each combination of values for the cohesion parameter $\lambda_c \in \{0.01, 0.1, 0.2\}$ and viscosity parameter $\lambda_v \in \{0.001, 10, 120\}$ (we fix all other constants).

To compare FleX and SPNets, we compute the intersection over union (IOU) of the two particle sets at each point in time. To compute the intersection between two particle sets with real valued cartesian coordinates, we relax the "identical" constraint to be within a small ϵ, i.e., the intersection is the set of all particles from each set that have at least one neighbor in the opposing set that is within ϵ units away. For this comparison we set ϵ to be 2.5cm. For the *scooping* scene, the IOU was 91.0%, and for the *ladle* scene, the IOU was 97.1%. Given this, it is clear that while SPNets is not identical to FleX, it matches closely and produces stable fluid dynamics.

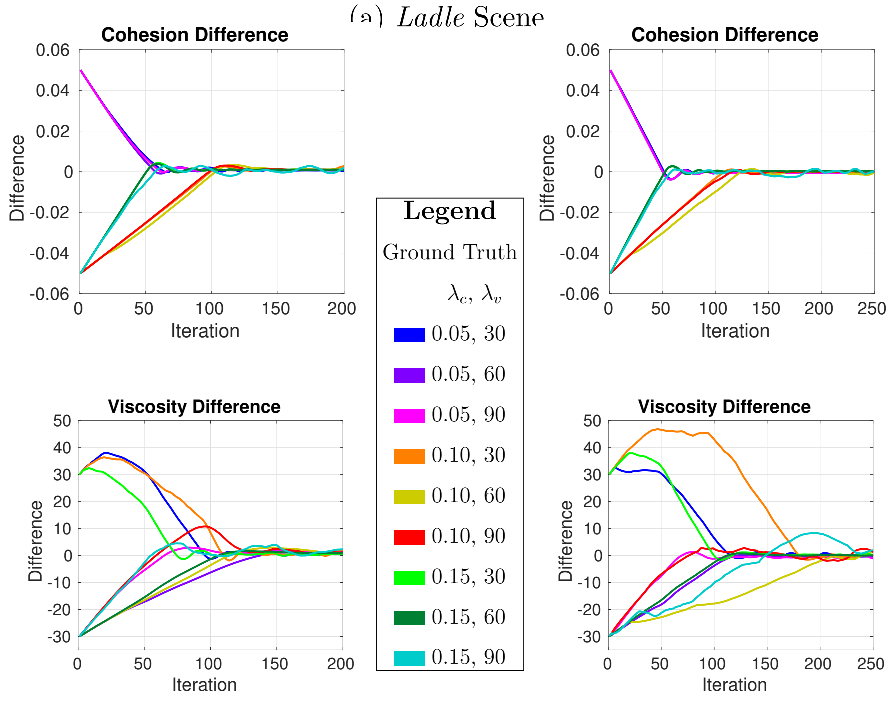
8.3.2 Estimating Fluid Parameters

We evaluate SPNets on the task of learning, or estimating, some of the λ_* fluid parameters from data. This experiment illustrates how one can perform system identification on liquids using gradient-based optimization. Here we frame this system identification problem as a learning problem so that we can apply learning techniques to discover the parameters. We use Nvidia FleX [91] to generate ground truth data, and then use backpropagation and gradient descent to iteratively update the parameter values until convergence. FleX is a commercially available physics simulation engine which implements fluid dynamics using Position Based Fluids (PBF).

Given sequences $\mathcal{P} = \{P_t\}$ and $\mathcal{V} = \{V_t\}$ of particle positions and velocities over time generated by FleX, at each iteration we do the following. First we randomly sample B particle positions P and velocities V from $\mathcal{P}$,$\mathcal{V}$ to make a training batch $\mathcal{P}_B$, $\mathcal{V}_B$. Next, SPNet is used to roll out the dynamics T timesteps forward in time to generate $\widetilde{\mathcal{P}}_{B+T}$, $\widetilde{\mathcal{V}}_{B+T}$, the predicted particle positions and velocities after T timesteps. We then compute the loss



(a) *Ladle* Scene



(b) L1-Loss

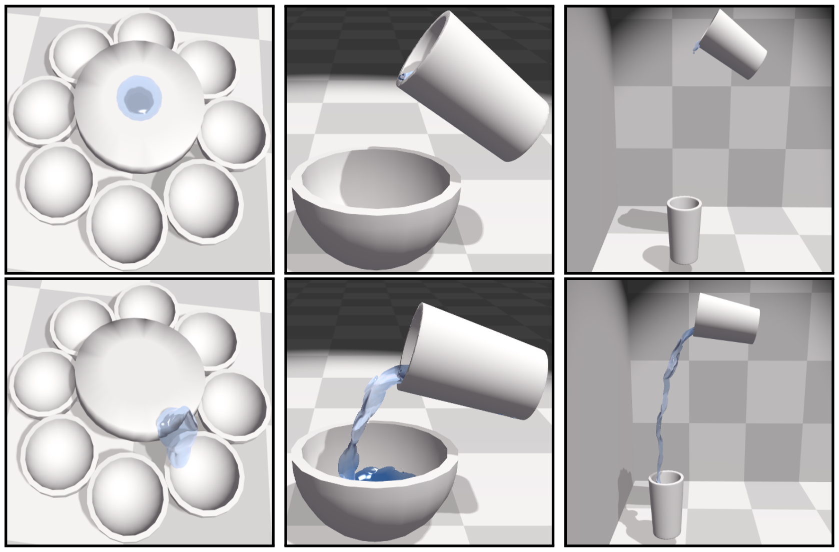
(c) Projection-Loss

Figure 8.3: (Top) The *ladle* scene. The left two images are before and after snapshots, the right image shows the particles projected onto a virtual camera image (with the objects shown in dark gray for reference). (Bottom) The difference between the estimated and ground truth fluid parameter values for cohesion λ_c and viscosity λ_v after each iteration of training for both the L1 loss and the projection loss. The color of the lines indicates the ground truth λ_c and λ_v values. The cohesion parameter λ_c was initialized to 0.1 in all cases except when the ground truth was 0.1, in which case it was initialized to 0.05. The viscosity parameter λ_v was initialized to 60 in all cases except when the ground truth was 60, in which case it was initialized to 30.

$l(\widetilde{\mathcal{P}}_{B+T}, \widetilde{\mathcal{V}}_{B+T}, \mathcal{P}_{B+T}, \mathcal{V}_{B+T})$ between the predicted positions and velocities and the ground truth positions and velocities. Since our model is differentiable, we can use backpropagation to compute the gradient of the loss with respect to each of the fluid parameters. We then take a gradient step to update the parameters. This process is repeated until the parameters converge.

We used the *ladle* scene shown in Figure 8.3a to test our method. Here, the liquid rests in a rectangular container as a ladle scoops some liquid from the container and then pours it back into the container. We generated 9 sequences, one for each combination of the cohesion parameter $\lambda_c \in \{0.05, 0.1, 0.15\}$ and viscosity parameter $\lambda_v \in \{30, 60, 90\}$ (we fixed all other fluid parameters). Each sequence lasted exactly 620 frames. We set our batch size to 8, T to 2, and use Adam [62] with default parameter values and a learning rate of 1e$-$2 to update the fluid parameters at each iteration. We evaluate using 2 different loss functions. The first is an L1 loss between the predicted and ground truth particle positions and velocities. This is possible because we know which particle in Flex corresponds to which particle in the SPNet prediction. In real world settings, however, such a data association is not known, so we evaluate a second loss function that eschews the need for it. We use the projection loss, which simulates a camera observing the scene and generating binary pixel labels as the observation (similar to the heatmaps generated by our method in chapter 4). We compute the projection loss between the predicted and ground truth states by projecting the visible particles onto a virtual camera image, adding a small Gaussian around the projected pixel-positions of each particle, and then passing the entire image through a sigmoid function to flatten the pixel values. The loss is then the L1 difference between the projected image of the predicted particles and the ground truth particles. Projecting the particles as a Gaussian allows us to compute smooth gradients backwards through the projection. For the *ladle* scene, the camera is placed horizontally from the ladle, looking at it from the side.

Figures 8.3b and 8.3c show the difference between the ground truth and estimated values for the cohesion λ_c and viscosity λ_v parameters when training the model on each of the 9 sequences. In all 9 cases and for both losses, the network converges to the ground truth



(a) *Plate* (b) *Pouring* (c) *Catching*

Figure 8.4: The control scenes used in the evaluations in this chapter. The top row shows the initial scene; the bottom row shows the scene after several seconds.

parameter values after only a couple hundred iterations. While the L1 loss tended to converge slightly faster (which is to be expected with a more dense loss signal), the projection loss was equally able to converge to the correct parameter values, indicating that the gradients computed by our model through the camera projection are able to correctly capture the changes in the liquid due to its parameters. Note that for the projection loss the camera position is important to provide the silhouette information necessary to infer the liquid parameters.

8.3.3 Liquid Control

To test the efficacy of the gradients produced by our models, we evaluate SPNets on 3 liquid control problems. The goal in each is to find the set of controls $\mathcal{U} = \{u_t\}$ that minimize the cost

$$L = \sum_t l(P_t, V_t, u_t) \tag{8.4}$$

where l is the cost function, P_t is the set of particle positions at time t, and V_t is the set of particle velocities at time t. P_t and V_t are defined by the dynamics as follows

$$P_t, V_t = SPN(P_{t-1}, V_{t-1}, OP(u_t)) \tag{8.5}$$

where SPN is the fluid dynamics computed by SPNets, and OP transforms the control u_t to the poses of the rigid objects in the scene at time t. The initial positions P_0 and velocities V_0 of the particles, the loss function l, and the control function OP are fixed for each specific control task.

To optimize the controls $\mathcal{U}$, we utilize Model Predictive Control (MPC) [19]. MPC optimizes the controls for a short, finite time horizon, and then re-optimizes at every timestep. Specifically, given the current particle positions P_t and velocities V_t and the set of controls $\mathcal{U}$ computed at the previous timestep, MPC first computes the future positions $P_{t+1}, ..., P_{t+T}$ and velocities $V_{t+1}, ..., V_{t+T}$ by repeatedly applying the SPNet for some fixed horizon T. Then, MPC sums the loss over this horizon as described in equation (8.4) and computes the gradient of the loss L with respect to each control Δu_t via our differentiable model. Finally, the updated controls $\mathcal{U}'$ are computed as follows

$$\mathcal{U}' = \left\{ u_i - s\frac{\Delta u_i}{\|\Delta u_i\|} \;\middle|\; i \in [t, t+T] \right\}$$

where s is a fixed step size. The first control $u'_t \in \mathcal{U}'$ is executed, the next particle positions P_{t+1} and velocities V_{t+1} are computed, and this process is repeated to update all controls again. Note that this process updates not only the current control u_t but also all controls in the horizon, so that by the time control u_{t+T} is executed, it has been updated T times. Figure 8.5 shows a diagram of this process. We set T to 10 and use velocity controls on our 3 test scenes. The controls u are initialized to 0 and T is set to a fixed horizon for each scene.

The *Plate* Scene: Figure 8.4a shows the *plate* scene. It consists of a plate surrounded by 8 bowls. A small amount of liquid is placed on the center of the plate, and the plate must

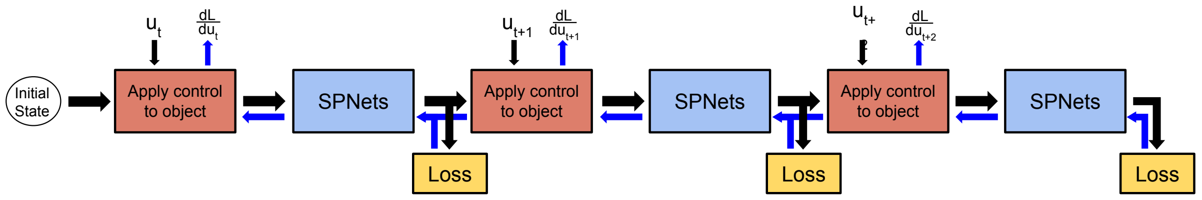


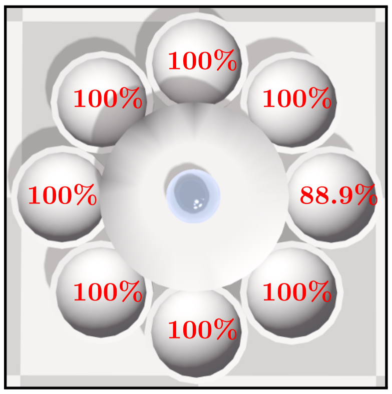
Figure 8.5: Diagram of the rollout procedure for optimizing the controls u. The dynamics are computed forward (black arrows) for a fixed number of timesteps into the future (shown here are 3). Then gradients of the loss are computed with respect to the controls backwards (blue arrows) through the rollout using backpropagation.

be tilted such that the liquid falls into a given target bowl. The controls for this task are the rotation of the plate about the x (left-right) and z (forward-backward) axes[3]. We set the loss function for this scene to be the L2 (i.e., Euclidean) distance between the positions of the particles and a point in the direction of the target bowl. We ran 8 evaluations on this scene, once with each bowl as the target.

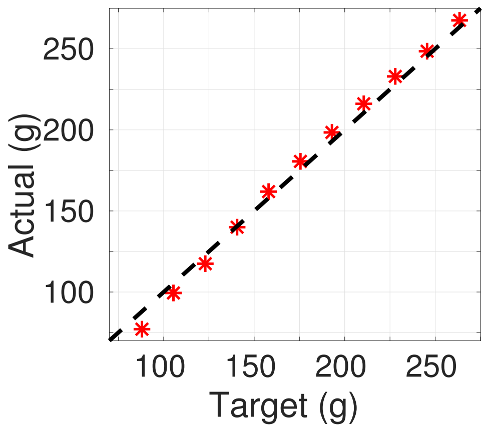
Figure 8.6a shows the results of each of the evaluations on the *plate* scene. In every case, the optimization produced a trajectory where the plate would "dip" in the direction of the target bowl, wait until the liquid had gained sufficient momentum, and then return upright, which allowed the liquid to travel further off the edge of the plate. Note that simply "dipping" the plate would result in the liquid falling short of the bowl. For all bowls except one, this resulted in 100% of the liquid being placed into the correct bowl. For the one bowl, when it was set as the target, all but a small number of the liquid particles were placed in the bowl. Those particles landed on the lip of the bowl, eventually rolling off onto the ground. Nonetheless, it is clear that our method is able to effectively solve this task in the vast majority of cases.

The *Pouring* Scene: We also evaluated our method on the *pouring* scene, shown in Figure 8.4b. The goal of this task is to pour liquid from the cup into the bowl. The control is

[3]In all our scenes, the y axis points up



(a) *Plate* Scene



(b) *Pouring* Scene

Initial Movement	MPC	Policy Train	Policy Test
Right	97.9%	98.3%	99.2%
Left	99.7%	99.5%	93.8%
Both	98.8%	98.9%	96.5%

(c) *Catching* Scene

Figure 8.6: Results from the liquid control task. From left to right and top to bottom: The *plate* scene. The numbers in each bowl indicate the percent of particles that were successfully placed in that bowl when that bowl was the target. The *pouring* scene. The x axis is the targeted pour amount and the y axis is the amount of liquid that was actually poured where the red marks indicate each of the 11 pours. The *catching* scene. Shown is the percent of liquid caught by the target cup where the rows indicate the initial direction of movement of the source.

the rotation of the cup about the z (forward-backward) axis, starting from vertical. Note that there is no limit on the rotation; the cup may rotate freely clockwise or counter-clockwise. Since the cup needs to perform a two part motion (turning towards the bowl to allow liquid to flow out, then turning back upright to stop the liquid flow), we use a two part piecewise loss function. For the first part, we set the loss to be the L2 distance between all the liquid particles and a point on the lip of the cup closest to the bowl. Once a desired amount of liquid has left the cup, we switch to the second part, which is a standard regularization loss, i.e., the loss is the rotation of the cup squared, which encourages it to return upright.

We ran 11 evaluations of this scene, varying the desired amount of poured liquid between 75g and 275g. In every case, all liquid either remained in the cup or was poured into the bowl; no liquid was spilled on the ground. For that reason, in figure 8.6b we show how close each evaluation was to the given desired pour amount. In every case, the amount poured was within 11g of the desired amount, and the average difference across all 11 runs between actual and desired was 5g. Note that the initial rotation of the cup happens implicitly; our loss function only specifies a desired target for the liquid, not any explicit motion. This shows that physical reasoning about fluids using our model enables solving fine-grained reasoning tasks like this.

The *Catching* Scene: The final scene we evaluated on was the *catching* scene, shown in Figure 8.4c. The scene consisted of two cups, a source cup in the air filled with liquid and a target cup on the ground. The source cup moved arbitrarily while dumping the liquid in a stream. The goal of this scene is to shift the target cup along the ground to catch the stream of liquid and prevent it from hitting the ground. The control is the x (left-right) position of the cup. In order to ensure smooth gradients, we set the loss to be the x distance between each particle and the centerline of the target cup inversely weighted by that particle's distance to the top of the cup. The source cup always rotated counter-clockwise (CCW), i.e., always poured out its left side.

We ran 8 evaluations of our model, varying the movement of the source cup. In every case, the source cup would initially move left/right, then after a fixed amount of time, would

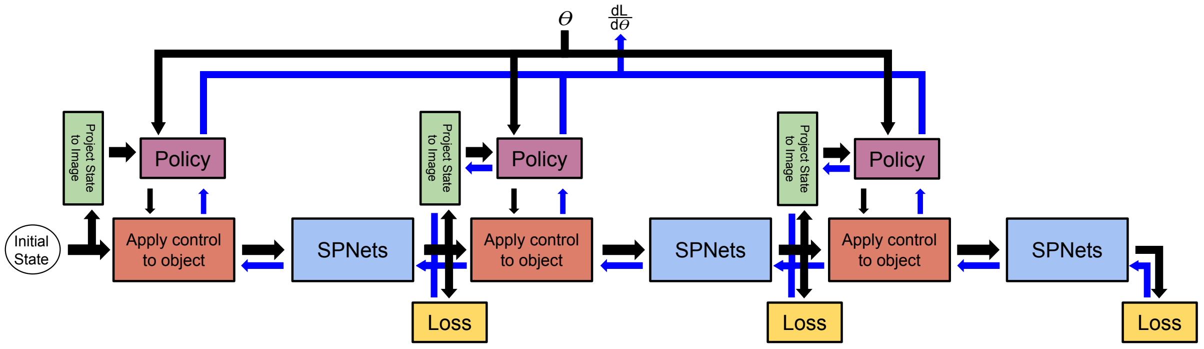


Figure 8.7: Diagram of the rollout procedure for optimizing the policy parameters θ. This is very similar to the procedure shown in Figure 8.5. The dynamics are computed forward (black arrows) for a fixed number of timesteps into the future (shown here are 3). Then gradients of the loss are computed with respect to θ backwards (blue arrows) through the rollout using backpropagation.

switch directions. Half the evaluations started with left movement, the other half right. We vary the switch time between 3.3s, 4.4s, 5.6s, and 6.7s. The first column of the table in Figure 8.6c shows the percentage of liquid caught by the cup. In all cases, the vast majority of the liquid was caught, with only a small amount dropped due largely to the time it took the target cup to initially move under the stream. It is clear from these results and the liquid control results on the previous two scenes that our model can enable fine-grained reasoning about fluid dynamics.

8.3.4 Learning a Liquid Control Policy via Reinforcement Learning

Finally, we evaluate our model on the task of learning a policy in a reinforcement learning setting. That is, the control u_t at timestep t is computed as

$$u_t = \pi(o_t, \theta)$$

where o_t is the observation at time t, θ are the policy parameters, and π is a function mapping the observation (and policy parameters) to controls. Since we have access to the full state, we compute the observation o_t as a function of the particle positions P_t and velocities V_t. The goal is to learn the parameters θ^* that best optimize a given loss function l.

To do this, we can use a technique very similar to the MPC technique which we described in the previous section. The main difference is that because the controls u_t are a function of the policy, we optimize instead the policy parameters θ. We rollout the policy for a fixed number of timesteps, compute the gradient of the policy parameters with respect to the loss, and then update the parameters. This is possible because our model is fully differentiable, so we can use backpropagation to compute the gradients backwards through the rollout. Figure 8.7 shows a diagram of this process.

We test our methodology on the *catching* scene. To train our policy, we use the data generated by the 8 control sequences from the previous section using MPC on the *catching* Scene. At each iteration of training, we randomly sample a different timestep t for each of the 8 sequences, then rollout the policy starting from the particle positions P_t and velocities V_t. We initialize the target cup X position by adding Gaussian noise to the X position of the target cup at time t in the training sequences. The observation is computed by projecting the particles onto a virtual camera image as described in section 8.3.2. The camera is positioned so that both cups are in its field of view. Its X position is set to be the same as the X position of the target cup, that is, the camera moves with the target cup.

Since the observation is effectively binary pixel labels, we use a relatively simple model to learn the policy. We use a convolutional neural network with 1 convolutional layer (10 3×3 kernels with stride of 2) followed by a rectified linear layer, followed by a linear layer with 100 hidden units, followed by another rectified linear layer, and finally a linear layer with 1 hidden unit. We feed the output through a hyperbolic tangent function to ensure it stays within a fixed range. We trained the policy for 1200 iterations using the Adam [62] optimizer with a learning rate of 2.5e$-$5. The input to the network is a 160×120 image and the output is the control.

The middle column of the table in figure 8.4c shows the percent of liquid caught by the target cup when using the policy to generate the controls for the same 8 sequences as were used in section 8.3.3 on the *catching* scene. In all 8 cases, the vast majority of the liquid was caught by the target cup. However, these were the same 8 sequences that the policy was trained on. To test the generalization ability of the policy, we modified the sequences as follows. For all the training sequences, the source cup rotated counter-clockwise (CCW) when pouring. To test the policy, we had the source cup follow the same X movement, but rotated clockwise (CW) instead, i.e., in training the liquid always poured from the left side of the source, but for testing it poured out of the right side. The percent of liquid caught by the target cup when using the policy for the CW case is shown in the third column of the table in figure 8.4c. Once again the policy is able to catch the vast majority of the liquid. The main point of failure is when the source cup initially moves to the left. In this case, as the source cup rotates, the liquid initially only appears in the upper-left of the image. It's not until the liquid has traveled downward several centimeters that the policy begins to move the target cup under the stream, causing it to fail to catch the initial liquid. This behavior makes sense, given that during training the policy only ever encountered the source cup rotating CCW, resulting in liquid never appearing in the upper-left of the image. Nonetheless, these results show that, at least in this case, our method enables us to train robust policies for solving liquid control tasks.

8.4 Combining Perception and SPNets

While the focus in this chapter has been on introducing SPNets as a platform for differentiable fluid dynamics, we also wish to show an example of how it can be combined with real perception. So in this section we show some results on a liquid state tracking task. Similar to the last chapter, we use SPNets to simulate liquid alongside real liquid and we use perception to close the loop. We combine SPNets with the perception methodology from chapter 4 to create a full deep network architecture for simulating and tracking liquids. We evaluate this on a set of pouring sequences collected on a real robot.

8.4.1 Task Definition

We task the robot with tracking the 3D state of liquid over time (this is the same task as in the previous chapter, chapter 7). We assume the robot has 3D mesh models of all the rigid objects in the scene and knows their pose at all points in time. We further assume that the robot knows the initial state of the liquid (e.g., the amount of liquid in a container). The robot then interacts with the objects and observes the scene with its RGBD camera. The task of the robot is to use its observations from its camera to track the changes in the 3D liquid state over time.

8.4.2 Methodology

To track the liquid state, the robot takes advantage of its knowledge of fluid dynamics built-in to SPNets. In an ideal world, knowing the initial state of the liquid and the changes in poses of the rigid objects, it should be possible to simulate the liquid alongside the real liquid, where the simulated liquid would perfectly track the real liquid. However, no model is perfect and there is inevitably going to be mismatch between the simulation and the real liquid. Furthermore, due to the temporal nature of this problem, a small error can quickly compound to a large deviation. The solution in this case is to "close the loop," i.e., use the robot's perception of the real liquid to correct the state of the simulation to prevent errors from compounding and better match the real liquid.

The methodology we adopt here is very similar to that of chapter 7 where the robot simulates the liquid forward in time alongside the real liquid while using perception to correct the state at each timestep. In this section, we use only pouring interactions, so we simulate each as follows. Each interaction starts with a known amount of liquid in the source container. The robot initializes the liquid state by placing a corresponding amount of liquid particles in the 3D mesh of the source container. Then, at each timestep, the robot updates the poses of the rigid objects and simulates the liquid forward for 1 timestep. During the simulation step, the robot uses its perception to correct the particle positions (described in the next paragraph). Each timestep corresponds to $\frac{1}{30}$ of a second in simulation time. The robot repeats this simulation process until the interaction is over.

The main difference between the methodology here and that in chapter 7 is that instead of assuming the robot has access to an expensive thermographic camera (and is using heated water), we assume the robot has access only to an inexpensive RGB camera. Thus we must use a different methodology than in chapter 7 to integrate the perception into the simulation. In chapter 4 we developed several deep network architectures for producing pixel-wise liquid labels from RGB images. Here we use the LSTM-FCN with RGB input to convert raw RGB images to pixel-wise liquid labels. We refer the reader to that chapter for more details on that network. We treat the perception correction as a constraint, similar to the pressure or cohesion constraints, allowing us to add it to the inner loop of the PBF algorithm (lines 5–9 in Figure 8.1). We define the function SolvePerception that takes as input the current set of particle locations P' and the RGB image R and produces ΔP^R, the vector to move each particle by to better satisfy the perception constraint. We insert this function immediately after line 7 and append ΔP^R to the summation on the following line of the algorithm. This adds the perception correction to be computed alongside the other corrections in the inner-loop of PBF. Note that because this is added as part of the inner loop in PBF, the velocity is automatically updated based on this correction on line 11.

Figure 8.8 shows a diagram of how we implement the SolvePerception function. We first apply the LSTM-FCN to the RGB image to generate pixel-wise liquid labels, which we refer to as *observed* labels. Next we compute the *observed* distance field over the *observed* labels, i.e., the distance from each pixel to the closest positive liquid pixel. We use this and a 2D convolution with fixed parameters to compute the *observed* distance field gradient, i.e., for every pixel the direction to the closest positive liquid pixel. In parallel, we project the state of the particles onto a 2D image using the camera's intrinsic and extrinsic parameters. This also generates a pixel-wise label image, which we refer to as the *model* labels. We then subtract the *model* labels from the *observed* labels to generate the *disparity* image, i.e., the pixels for which the *model* and *observation* disagree. We then generate the *disparity* distance

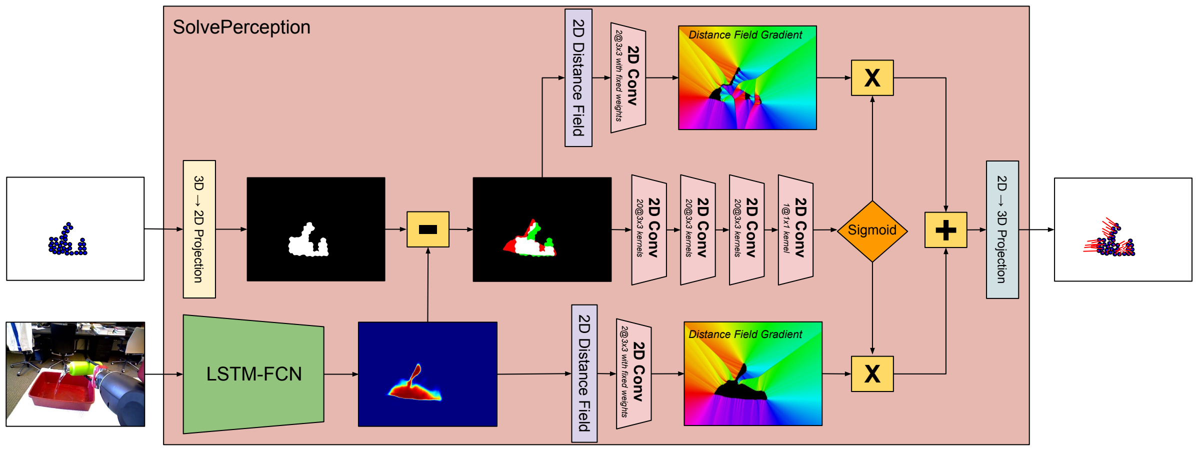


Figure 8.8: A diagram of the SolvePerception method. It takes as input the current particle state (center-left) and the current RGB image (lower-left). The RGB image is passed through the LSTM-FCN to generate pixel labels, the particle state is projected onto 2D. From this, 2 gradient fields are computed, where the gradient points in the direction of the closest *liquid* pixel. These fields are then blended and projected onto the particles in 3D.

field gradient in the same manner as for the *observed* distance field gradient. Furthermore, we feed the *disparity* image through 4 2D convolutions (the first three of which are followed by a ReLU) and a sigmoid function. The output of this is a blending value for each pixel. These blending values determine how to combine the two distance field gradients. We multiply the *observed* distance field gradient by the blending values, and the *disparity* distance field gradient by 1 minus the blending values and add them together. This results in a blended gradient field. We project this back onto the particles in 3D, adjusting the gradient by the camera parameters. The result is the set ΔP^R, the distance to move each particle to better match the perception. Note that for both projections (projecting 3D particles onto the 2D image plane and projecting the 2D gradient field onto the 3D particles), we ignore particles that are blocked from view of the camera by an object (e.g., particles in a container).

To train the parameters of the network, we do the following. We train the parameters of the LSTM-FCN part of the network using the same training data (holding out our test data from the next section) and methodology as in chapter 4 and we refer the reader there for details. We then fix those parameters for the remainder of the training. We pre-train the entire network in Figure 8.8 by randomly selecting frames from our dataset and randomly placing particles in the scene. To compute the loss, we use the ground truth pixel labels collected from the thermal camera as described in chapter 4. The loss is computed as

$$L(\mathcal{L}, P) = \left(\sum_{p \in P} \min_{l \in \mathcal{L}} ||PROJ(p) - l||^2 \right) + \left(\sum_{l \in \mathcal{L}} \min_{p \in P} ||l - PROJ(p)||^2 \right)$$

where $\mathcal{L}$ is the set of positive liquid pixels in the ground truth image, P is the set of particle positions, and $PROJ$ projects a particle location from 3D onto the 2D image plane. Intuitively, this loss computes 2 terms: the *accuracy*, i.e., how far each particle is from a liquid pixel, and the *coverage*, i.e., how far each liquid pixel is from a particle or how well the particles cover the liquid pixels. We pre-train this network for 48,000 iterations using ADAM [62] with a learning rate of 0.0001, default momentum values, and a batch size of 4. Finally, we train the network from end-to-end by adding SolvePerception into SPNets, unrolling it over time, and computing the same loss. Again, this is possible because SPNets can propagate the gradient backwards in time from one timestep to the previous, allowing us to use those gradients to update the learned weights. We trained the network this way for 3,500 iterations also using ADAM with the same learning rate and momentum values, a batch size of 1, and unrolling for 30 timesteps[4]. A diagram showing this training process is shown in Figure 8.9.

8.4.3 Evaluation & Results

We evaluated SPNets combined with perception on the data from the *pouring* test set in chapter 7. That is, the robot executed 12 pouring sequences following a fixed trajectory, half with the *cup* and half with the *bottle*. One third of the sequences the container started 30% full, one third 60% full, and one third 90% full. We tracked the liquid state using

[4]Unrolling the network for training here is the same unrolling technique we used to train the LSTM-FCN in chapter 4.

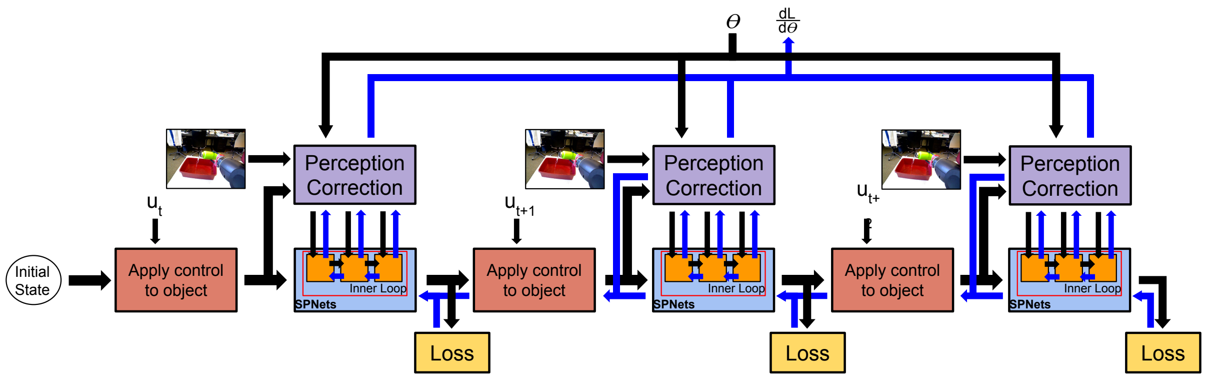
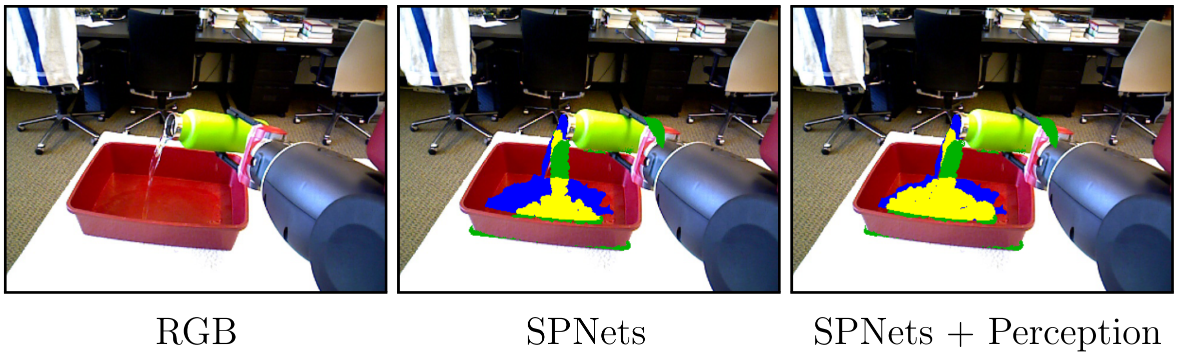
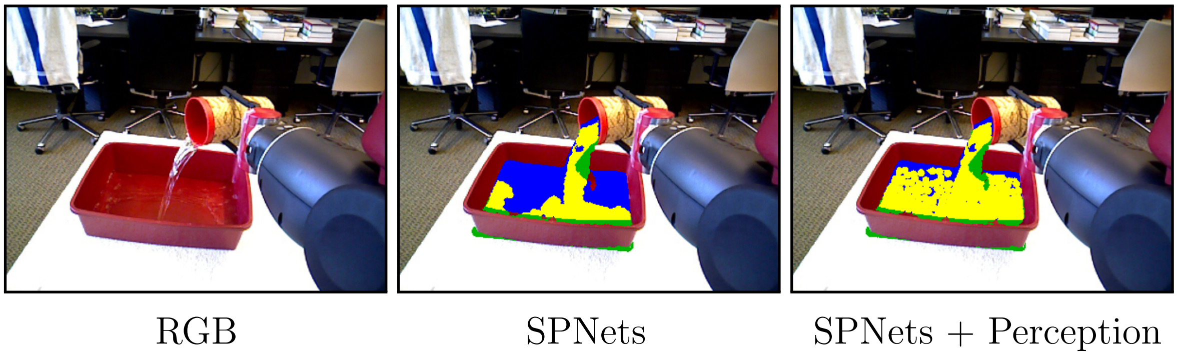


Figure 8.9: Diagram of the rollout procedure for optimizing the parameters of the perception network θ. This is very similar to the procedures shown in Figures 8.5 and 8.7. The dynamics are computed forward (black arrows) for a fixed number of timesteps into the future (shown here are 3). In this case the controls u are fixed. The gradients of the loss are computed with respect to θ backwards (blue arrows) through the rollout using backpropagation. The box "Perception Correction" corresponds to the network shown in Figure 8.8 and θ are its parameters. Note that here the inner loop of the PBF algorithm in SPNets is shown (with 3 iterations) since this is how the perception correction is intergrated into the dynamics.



(a) Example Frame for the *bottle*



(b) Example Frame for the *cup*

Figure 8.10: Results when combining SPNets with perception. The images in the top and bottom rows show 2 example frames. From left-to-right: the RGB image (for reference), the RGB image with SPNets overlayed, and the RGB image with SPNets with perception overlayed. In the overlays, the blue color indicates ground truth liquid pixels, green indicates the liquid particles, and yellow indicates where they overlap.

both SPNets with perception and SPNets alone for comparison. For each sequence, the known amount of liquid was placed in the source container at the start, and as the object poses were updated at each timestep, the liquid was also updated via SPNets. For SPNets with perception, SolvePerception was added to the PBF algorithm as described in the previous section and the RGB image from the robot's camera was used at each timestep as input to that function. For SPNets alone, SolvePerception was not added and the liquid state was tracked open-loop instead.

Figure 8.10 shows 2 example frames from the test set and the result of both SPNets and SPNets with perception. The yellow pixels indicate where the model and ground truth agree; the blue and green where they disagree. From this it is apparent that SPNets with perception is significantly better at matching the real liquid state than SPNets without perception. We evaluate the intersection-over-union (IOU) across all frames of the 12 pouring sequences. To compute the IOU, we compare the *true* pixel labels from the ground truth with the *model* pixel labels. To get the *model* pixel labels, we project each particle in the simulation onto the 2D image plane. However, since there are far fewer particles than pixels, we draw a circle of radius 5 around each particle's projected location. The result is a set of pixel labels corresponding to the state of the model. To compute the IOU, we divide the number of pixels where the *model* and *true* labels agree by the number of pixels that are positive in either set of labels.

SPNets alone achieved an IOU of 36.1%. SPNets with perception achieved an IOU of 56.8%. Note that unlike in chapter 7, we did not spend a significant amount of time fine-tuning the simulation parameters for either SPNets with or without perception. Thus these results clearly show that perception is capable of greatly improving performance even when there is significant model mismatch. Here we can see that SPNets with perception increased performance by 20%, and from the images in Figure 8.10 it is clear that this increase in performance is significant. This shows that our framework can be very useful for combining real perceptual input with fluid dynamics.

8.5 Discussion

In this chapter we presented SPNets, a method for computing differentiable fluid dynamics and their interactions with rigid objects inside a deep network. To do this, we developed the ConvSP and ConvSDF layers, which allow the model to compute particle-particle and particle-rigid object interactions. We then showed how these layers can be combined with standard neural network layers to compute fluid dynamics. Our evaluation in Section 8.3 showed how a fully differentiable fluid model can be used to 1) learn, or identify, fluid parameters from data, 2) control liquids to accomplish a task, and 3) learn a policy to control liquids. This is the power of model-based methods: they are widely applicable to a variety of tasks. Combining this with the adaptability of deep networks, we can enable robots to robustly reason about and manipulate liquids. Importantly, SPNets make it possible to specify liquid identification and control tasks in terms of the *desired state of the liquid*; the resulting controls follow from the physical interaction between the liquid and the controllable objects. This is in contrast to prior approaches to pouring liquids, for instance, where the relationships between controls and liquid states have to be specified via manually designed functions.

In this chapter, we proposed a method for combining the deep learning techniques used in earlier chapters of this thesis with the physics-based model of the prior chapter. Both methods proved to be powerful tools for enabling robots to interact intelligently with liquids. With SPNets, we showed how it can be applied to a variety of different tasks like the model-based methods of the previous chapter and how it can be used to adapt the model similar to the learning-based methods earlier in this thesis. This can enable robots to manipulate liquids robustly and precisely.

8.5.1 Practical Takeaways

While implementing SPNets, there were many practical implementation details that gave us interesting technical insights. One technical issue that we dealt with was the issue of particle clipping, which we also had to deal with in chapter 7. This is the issue where when object's have thin walls (such as a cup or a bottle), a particle's update vector from one timestep to the next may move it directly through that wall without ever stopping inside it, resulting in no detection of the particle-object interaction that should have occurred. Since SPNets represents rigid objects as SDFs and not triangle meshes like in chapter 7, we had to develop a new methodology to handle this. To fix this issue, anytime a particle's position is updated, we split the update vector into smaller pieces and check for collision at each position along that vector. Assuming the length of each piece is small enough, this means that we can determine if a particle will collide with a rigid object and prevent them from clipping through the sides of the objects.

Another technical issue we encountered was computing the particle-particle interactions via the ConvSP layer efficiently. The naive way to compute the ConvSP equation is to iterate over every pair of particles and test if they are close enough to interact with each other. However the run time for this is quadratic in the number of particles. To make these pairwise tests more efficient, we utilize a hashgrid. The hashgrid divides up the 3D space into a grid and then makes a list of all particles that fall in each cell. Now to compute the ConvSP equation, we only need to test the particles in nearby grid cells to each particle, rather than iterate over every pair. This algorithm is linear time in the number of particles, assuming that the number of particles in each grid cell is relatively constant. However, when we implemented this hashgrid in our ConvSP code, for approximately 10k particles, it was slower than the naive method! After careful analysis, we discovered the reason. Our ConvSP layer is implemented on a graphics card using Cuda. The graphics card has both global memory, similar to system memory on a computer, and a cache for each Cuda core, similar to the CPU cache on most processors. Importantly, accessing data stored in global memory is slow, but accessing data stored in the cache is much faster, so efficient Cuda code attempts to maximize the number of cache hits. However, when iterating over the particles in a grid cell, their data would be stored in the original particle list in global memory, and so the particles in the same grid cell would essentially be randomly spread in the data for all particles. Thus many of the data accesses for the particles in a grid cell would cause a cache miss, requiring the computation to halt while slower global memory was accessed. To fix this, after computing the hashgrid, we reorder all the data such that all particles in the same cell are contiguous in memory. This resulted in many fewer cache misses, and made our hashgrid method much faster, even faster than the naive method. This is the kind of insight that is often not stated in research like this, however it turned out to be fundamentally important to the efficiency of our ConvSP layer.

Chapter 9
CONCLUSION

9.1 Summary

In this thesis we investigated methods for robots to *perceive*, *reason about*, and *manipulate* liquids. In chapter 1 we motivated our investigation with an example of a household task: baking a cake. There, we noted how many of the prerequisite skills for solving that task and other similar household tasks are already under investigation, but missing from this is an investigation into how to handle the liquids required for such a task. So we focused this thesis on looking at ways to fill this knowledge gap. In chapter 2 we discussed what work had already been done in this area, and how our work related.

We began our investigation in chapter 3 by starting with a common liquid task: pouring. To focus on the task at hand, we simplified the problem by placing a real-time scale underneath the target container. The robot was then given a source container containing an unknown amount of liquid and tasked with pouring a specific amount out into the target. Using Guided Policy Search, the robot was able to learn an effective policy to pour precise amounts of liquid. We found that in all cases, the robot was able to achieve less than 10ml of error, an amount on par with what would be expected of a human performing the same task. This showed that with the right state information, it is possible for a robot to precisely control liquids.

In the following chapter, chapter 4, we relaxed one of the simplifying assumptions from chapter 3. We removed the real-time scale and focused on perceiving the liquid from raw sensor data, in this case a color camera. We trained several different deep neural networks to examine their performance on solving the *detection* task, i.e., labelling pixels in an image as containing liquid or not containing liquid. Our results showed that integrating temporal information alongside image features was necessary to solve this task with a high degree of accuracy. This aligned well with our intuition as clear liquids, like the water used in these experiments, is difficult to see from still images, but can more readily be seen by monitoring changes in reflections and refractions over time.

In chapter 5 we combined work from the previous two chapters to revisit the pouring task. However, this time we removed the real-time scale and instead required the robot to rely on its color camera to perceive the state of the liquid. We trained a deep network to convert the pixel-wise liquid detections of the network from the previous chapter into an estimate of the volume contained in the target container. By combining the two networks together, the robot was able to pour precise amounts of liquid. Using only RGB feedback, the robot achieved an average error of 38ml, which is on par with what a human would be expected to do under similar circumstances. This showed that precise control tasks are feasible even when done from raw perceptual data.

In the following chapter, chapter 6, we examined a reasoning task. Specifically, we looked at the task of *tracking*, which tasks the robot with finding all the liquid in a scene (visible and occluded), given a perceptual system capable of producing semantic labels for visible objects. We again tested several deep network architectures on this task, and similar to the perception results from chapter 4, we found that recurrent networks performed the best at this task. Not only could they learn basic liquid rules (e.g., liquid exiting from the lip of a container meant that the container was full below that point) but they could also remember where liquid had been seen previously (e.g., liquid disappeared into a container, so now that container has liquid in it).

However, the limitation of this methodology was that it was restricted to reasoning in 2D. So in chapter 7 we shifted to a fully 3D methodology. We investigated methods for using physics-based fluid dynamics models to perform reasoning tasks with liquids. We looked at how simple perceptual models could be used to "close the loop" between a robot's internal simulation of the liquid and the real liquid. Using this, the robot was able to reason about unknown state variables in the environment, such as how much liquid was contained in a container or where a potential obstacle was. The use of generic physics models showed the power of model-based methods: they could easily generalize from one task to another completely different task.

In chapter 8 we introduced SPNets, an implementation of fluid dynamics using deep learning tools. This allowed for not only a differentiable fluid simulator, but also easy integration with deep networks. We showed how this model could be used to learn fluid parameters such as cohesion or viscosity. We also showed how it can be used to solve different liquid control tasks and to train a reinforcement learning policy directly using policy gradients. Finally, we showed how SPNets can be combined with the perception network from chapter 4 to track the state of the real liquid accurately.

9.2 Discussion

In chapter 1 we laid out the problem statement for this thesis. We described 3 subproblems necessary to solve for robots to intelligently interact with liquids: *perceiving* liquids, *reasoning about* liquids, and *manipulating* liquids. Let us examine here what we learned about them in this thesis.

Perceiving Liquids: The perception of liquids was the primary focus of chapter 4. In that chapter, we discovered that perceiving transparent liquids like water required some ability to integrate information over time; it wasn't enough to simply look at static images, the robot needed to see the motion of the liquid to really perceive it. This aligns with intuition. Due to their non-rigidity, liquids can appear in an exponential number of configurations, making it difficult to deduce the location of liquid from only a few features. Additionally, the transparency of the liquids can further decrease the amount of information that can be gathered about the liquid from a single image. Thus by adding more images with motion, and thus more information, it becomes possible to perceive liquids accurately. Our recurrent neural network performed the best at this task. It could pass information forward in time, incrementally updating its perception from one frame to the next, making the problem much easier than solving it from scratch each frame. This shows that passing forward in time some notion of state can make perceiving high dimensional substances such as liquids easier.

Reasoning About Liquids: In chapters 6 and 7 we examined ways for a robot to *reason about* liquids. The results of chapter 6 showed how a robot could use deep neural networks to learn basic, intuitive physical rules about liquids from data, such as if liquid is unsupported it falls or if liquid exits from the lip of a container then that container is full up to the lip. Interestingly, even the non-recurrent networks, that did not contain any state information, were able to learn some of these rules, albeit not ones that requires memory such as if liquid disappears from sight into a container, it remains in the container. This showed that some liquid physics can be inferred from the relationships between liquid and objects around it. We extended our methodology to 3D in chapter 7. There we showed how our physics-based model could be applied to two entirely different tasks. This really exemplified the power of model-based methods: they can easily generalize between tasks. What this showed was that model-based methods like this can be applied to a wide variety of tasks, something which may be difficult for a model-free method to do. Due to the high-dimensionality of liquids, it is easy for model-free methods to overfit to their training data, making generalization difficult. In contrast, our results show that our model-based method can counteract this tendency to overfit by incorporating physics knowledge into the model.

Manipulating Liquids: We examined manipulating liquids in chapters 3, 5, and 8. In chapters 3 and 5, we found that the robot could pour specific amounts of liquid with precision given a good, task-relevant summarization of the liquid state (e.g., the estimate of volume of liquid in the target container). This showed that precise control of liquids is possible, and that learning-based methods are well-suited to learning a specific task. In chapter 8 we used our model SPNets to examine several control tasks. Our model was able to control the liquid using the full state representation. This shows that while liquid control may present a high-dimensional, challenging task, the use of physics-based models can make the task more feasible. Liquid control is challenging, and using end-to-end model-free methods may not be feasible to do so in a general setting. Our results here show that this is not the case for methods that use strong physics priors; they are able to accomplish this challenge in at least a limited sense without requiring huge amounts of training data.

In this thesis we also developed several systems that helped our robot to interact with liquids:

- **Acquiring Ground Truth Labels with a Thermographic Camera:** One system we developed in this thesis was a method for acquiring ground truth labels in a real robot environment. We used a thermographic camera calibrated to an RGB camera combined with heated water. The heated water, being much warmer than its surroundings, appeared very bright in the thermal image, making it simple to threshold the values and acquire pixel-wise liquid labels. To the best of our knowledge, we are the first to use this method.

- **Application of Deep Learning to Liquid Perception:** In chapter 4 we developed 3 deep neural network architectures for perceiving liquids, one that saw only static images of the scene, one that saw short windows of movement, and one with an explicit memory. For all of them one challenge we dealt with was the large imbalance between *liquid* and *not-liquid* pixels. We overcame this by first pre-training the networks on crops of images focused around liquid, alleviating some of the imbalance, and also by weighting the loss of *not-liquid* pixels by 0.1. Furthermore, we overcame the exploding/vanishing gradient problem by introducing an LSTM layer into our recurrent network, which then enabled it to integrate information over time, making the perception task more feasible. The LSTM-FCN that we developed was able to perceive liquids with a high degree of accuracy.

- **Creation of a Closed-Loop Liquid Simulator:** In chapter 7 we developed a closed-loop liquid simulator. We implemented fluid dynamics on top of a particle interaction library, incorporated rigid object tracking from depth cameras, and created 2 methods for using perceptual feedback to correct deviations in the liquid state. Furthermore, the system was implemented with graphics processor support, allowing it to run efficiently even with a high-dimensional liquid state. With this system, the robot was able to reason effectively about liquid in its environment. It could track the state of the liquid and then use that to infer unknown variables, such as the location of a hidden obstruction or the amount of liquid in a container.

- **Development of SPNets:** Finally, in chapter 8, we created SPNets, an implementation of fluid dynamics using the deep learning toolbox. This resulted in a fully differentiable fluid dynamics model. We showed how this differentiability can be used to infer liquid parameters from data, perform control tasks, and even train policies. We implemented it in the common deep learning library PyTorch, which makes it easy to incorporate it into other deep networks, which we did when we combined it with our perception network. We also released the code to allow other researchers to use it for research on liquids.

In this thesis, we investigated ways for robots to *perceive*, *reason about*, and *manipulate* liquids. We found that *perception* requires temporal integration, *reasoning* can be done robustly using physics-based models, and *control* can be done precisely given a precise state representation. We developed several systems from deep neural network architectures to differentiable fluid dynamics. The insights we gained from our investigation here shed light on how robots can intelligently interact with liquids, and the tools developed can enable robots to execute some of those insights. While there will always be more work to be done, in this thesis we help to bridge the gap in knowledge, bringing us one step closer to the cake baking robot that motivated this work to begin with.

9.3 Future Work

This thesis was an initial investigation into how robots can interact with liquids intelligently. However there are still plenty of avenues for continued work in this area. One such area is examining granular media such as flour or sand. Granular media have many of the same problems as liquids (high-dimensional state space, non-rigidity) and behave in similar ways, so it is a natural extension of this work. In fact, we have already done some work in this area [134]. But there is still more work that can be done, such as learning low-level action controllers or even incorporating granular media physics models into SPNets.

Another avenue for future work is to examine liquids other than water. For most of the work in this thesis, we use water in our liquid experiments. This is largely due to the ubiquity of water in human environments and due to it exemplifying many of the perception challenges that are common amongst liquids (e.g., transparency). We briefly examined other liquids in chapter 8 when we looked at finding liquid parameters from data, however this was entirely in simulation. Other liquids, such as oil or dough, appear frequently in common tasks in human environments, and it would be useful to investigate how our methods here would extend to those liquids. One challenge in doing so, though, would be acquiring ground truth labels. In chapters 4, 5, and 7 we used a thermal camera with hot water to acquire pixel-wise labels. However with other liquids, such as oil, heating them may cause them to behave differently than they would at room temperature. Thus in order to extend to other liquids, other methods of acquiring ground truth must be devised.

Finally, another avenue for future work is to examine how to apply long-horizon planning algorithms to liquid manipulation tasks. One difficulty in doing so is the high-dimensional state space. In chapters 3 and 5 we reduced the dimensionality of the state space by extracting task specific variables (the volume of liquid in the target container), which made the control tasks more feasible. This is not suitable for general-purpose planning though as it limits the tasks that can be performed. In chapter 8 we evaluated control tasks using the full dimensionality of the liquid, although in that case the tasks were short-horizon tasks. Future work will likely need to develop a method to summarize the state of the liquid that both captures enough information to accurately represent it in a relatively general manner but is also low-dimensional enough to allow for efficient planning.

BIBLIOGRAPHY

[1] Matlab statistics and machine learning toolbox. `http://www.mathworks.com/help/stats/index.html`. Accessed: 2015-12-31.

[2] David J Acheson. *Elementary fluid dynamics*. Oxford University Press, 1990.

[3] Nadir Akinci, Markus Ihmsen, Gizem Akinci, Barbara Solenthaler, and Matthias Teschner. Versatile rigid-fluid coupling for incompressible sph. *ACM Transactions on Graphics (TOG)*, 31(4):62, 2012.

[4] Rachid Alami, Jean-Paul Laumond, and Thierry Siméon. Two manipulation planning algorithms. In *WAFR Proceedings of the workshop on Algorithmic foundations of robotics*, pages 109–125. AK Peters, Ltd. Natick, MA, USA, 1994.

[5] Kostas Alexis, George Nikolakopoulos, and Anthony Tzes. Switching model predictive attitude control for a quadrotor helicopter subject to atmospheric disturbances. *Control Engineering Practice*, 19(10):1195–1207, 2011.

[6] Ethem Alpaydin. Reinforcement learning. In *Introduction to machine learning*, chapter 18, pages 517–290. MIT Press, 2014.

[7] Jeffrey L Bada. How life began on earth: a status report. *Earth and Planetary Science Letters*, 226(1-2):1–15, 2004.

[8] Christopher J Bates, Ilker Yildirim, Joshua B Tenenbaum, and Peter W Battaglia. Humans predict liquid dynamics using probabilistic simulation. In *Proceedings of the 37th annual conference of the cognitive science society*, 2015.

[9] Peter W Battaglia, Jessica B Hamrick, and Joshua B Tenenbaum. Simulation as an engine of physical scene understanding. *Proceedings of the National Academy of Sciences*, page 201306572, 2013.

[10] Dmitry Berenson, Siddhartha S Srinivasa, Dave Ferguson, and James J Kuffner. Manipulation planning on constraint manifolds. In *Robotics and Automation, 2009. ICRA'09. IEEE International Conference on*, pages 625–632. IEEE, 2009.

[11] Blender Online Community. *Blender - A 3D modelling and rendering package.* Blender Foundation, Blender Institute, Amsterdam, 2016.

[12] Phil Blunsom. Hidden markov models. *Lecture notes, August*, 15:18–19, 2004.

[13] Kenneth Bodin, Claude Lacoursiere, and Martin Servin. Constraint fluids. *IEEE Transactions on Visualization and Computer Graphics*, 18(3):516–526, 2012.

[14] Mario Bollini, Jennifer Barry, and Daniela Rus. Bakebot: Baking cookies with the pr2. In *The PR2 Workshop: Results, Challenges and Lessons Learned in Advancing Robots with a Common Platform, IROS*, 2011.

[15] Michael Bowling and Manuela Veloso. Multiagent learning using a variable learning rate. *Artificial Intelligence*, 136(2):215 – 250, 2002.

[16] Robert Bridson. *Fluid simulation for computer graphics.* CRC Press, 2015.

[17] Roxana Bujack and Kenneth I Joy. Lagrangian representations of flow fields with parameter curves. In *Large Data Analysis and Visualization (LDAV), 2015 IEEE 5th Symposium on*, pages 41–48. IEEE, 2015.

[18] Maya Cakmak and Andrea L Thomaz. Designing robot learners that ask good questions. In *ACM/IEEE International Conference on Human-Robot Interaction (HRI)*, pages 17–24, 2012.

[19] Eduardo F Camacho and Carlos Bordons Alba. *Model predictive control.* Springer Science & Business Media, 2013.

[20] Zhe Cao, Yaser Sheikh, and Natasha Kholgade Banerjee. Real-time scalable 6dof pose estimation for textureless objects. In *Robotics and Automation (ICRA), 2016 IEEE International Conference on*, pages 2441–2448. IEEE, 2016.

[21] Nilanjan Chakraborty, Stephen Berard, Srinivas Akella, and Jeffrey C Trinkle. A geometrically implicit time-stepping method for multibody systems with intermittent contact. *The International Journal of Robotics Research*, 33(3):426–445, 2014.

[22] Alexandre Joel Chorin. Numerical solution of the navier-stokes equations. *Mathematics of computation*, 22(104):745–762, 1968.

[23] Simon Clavet, Philippe Beaudoin, and Pierre Poulin. Particle-based viscoelastic fluid simulation. In *Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 219–228. ACM, 2005.

[24] Marc Deisenroth and Carl E Rasmussen. Pilco: A model-based and data-efficient approach to policy search. In *Proceedings of the 28th International Conference on machine learning (ICML)*, pages 465–472, 2011.

[25] Marc Peter Deisenroth, Dieter Fox, and Carl Edward Rasmussen. Gaussian processes for data-efficient learning in robotics and control. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 37(2):408–423, 2015.

[26] Chau Do, Tobias Schubert, and Wolfram Burgard. A probabilistic approach to liquid level detection in cups using an rgb-d camera. In *Proceedings of the International Conference on Intelligent Robots and Systems (IROS)*, pages 2075–2080, 2016.

[27] Christof Elbrechter, Jonathan Maycock, Robert Haschke, and Helge Ritter. Discriminating liquids using a robotic kitchen assistant. In *Proceedings of the International Conference on Intelligent Robots and Systems (IROS)*, pages 703–708, 2015.

[28] Tom Erez, Kendall Lowrey, Yuval Tassa, Vikash Kumar, Svetoslav Kolev, and Emanuel Todorov. An integrated system for real-time model predictive control of humanoid robots. In *Humanoid Robots (Humanoids), 2013 13th IEEE-RAS International Conference on*, pages 292–299. IEEE, 2013.

[29] Clement Farabet, Camille Couprie, Laurent Najman, and Yann LeCun. Learning hierarchical features for scene labeling. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 35(8):1915–1929, 2013.

[30] Vlad Firoiu, William F. Whitney, and Joshua B. Tenenbaum. Beating the world's best at super smash bros. with deep reinforcement learning. *arXiv preprint arXiv:1702.06230*, 2017.

[31] David A Forsyth and Jean Ponce. *Computer vision: a modern approach.* Prentice Hall Professional Technical Reference, 2002.

[32] Dieter Fox, Wolfram Burgard, and Sebastian Thrun. The dynamic window approach to collision avoidance. *IEEE Robotics & Automation Magazine*, 4(1):23–33, 1997.

[33] Felix Franks. *Water: a matrix of life.* Royal Society of Chemistry, 2007.

[34] Dhiraj Gandhi, Lerrel Pinto, and Abhinav Gupta. Learning to fly by crashing. In *International Conference on Intelligent Robots and Systems (IROS)*, pages 3948–3955. IEEE, 2017.

[35] Cristina Garcia Cifuentes, Jan Issac, Manuel Wüthrich, Stefan Schaal, and Jeannette Bohg. Probabilistic articulated real-time tracking for robot manipulation. *IEEE Robotics and Automation Letters (RA-L)*, 2016.

[36] Xiaoyu Ge, Jae Hee Lee, Jochen Renz, and Peng Zhang. Hole in one: Using qualitative reasoning for solving hard physical puzzle problems. In *ECAI*, pages 1762–1763, 2016.

[37] Robert A Gingold and Joseph J Monaghan. Smoothed particle hydrodynamics: theory and application to non-spherical stars. *Monthly notices of the royal astronomical society*, 181(3):375–389, 1977.

[38] Ross Girshick, Jamie Shotton, Pushmeet Kohli, Antonio Criminisi, and Andrew Fitzgibbon. Efficient regression of general-activity human poses from depth images. In *Computer Vision (ICCV), 2011 IEEE International Conference on*, pages 415–422. IEEE, 2011.

[39] Hoover William Graham. *Smooth Particle Applied Mechanics: The State Of The Art*, volume 25. World Scientific, 2006.

[40] Fabien Gravot, Atsushi Haneda, Kei Okada, and Masayuki Inaba. Cooking for humanoid robot, a task that needs symbolic and geometric reasonings. In *Robotics and Automation, 2006. ICRA 2006. Proceedings 2006 IEEE International Conference on*, pages 462–467. IEEE, 2006.

[41] Klaus Greff, Rupesh K Srivastava, Jan Koutník, Bas R Steunebrink, and Jürgen Schmidhuber. Lstm: A search space odyssey. *IEEE transactions on neural networks and learning systems*, 2016.

[42] S. Griffith, J. Sinapov, V. Sukhoy, and A. Stoytchev. A behavior-grounded approach to forming object categories: Separating containers from noncontainers. *IEEE Transactions on Autonomous Mental Development*, 4(1):54–69, 2012.

[43] S. Griffith, V. Sukhoy, and A. Stoytchev. Using sequences of movement dependency graphs to form object categories. In *11th IEEE-RAS International Conference on Humanoid Robots*, pages 715–720, 2011.

[44] Shane Griffith, Vladimir Sukhoy, Todd Wegter, and Alexander Stoytchev. Object categorization in the sink: Learning behavior–grounded object categories with water. In *Proceedings of the 2012 ICRA Workshop on Semantic Perception, Mapping and Exploration*. Citeseer, 2012.

[45] Tatiana López Guevara, Rita Pucci, Nicholas K Taylor, Michael Gutmann, Subramanian Ramamoorthy, and Kartic Subr. To stir or not to stir: Online estimation of liquid properties for pouring actions. In *Proceedings of Robotics: Science and Systems (RSS) Workshop on Learning and Inference in Robotics: Integrating Structure, Priors and Models*, 2018.

[46] Tatiana López Guevara, Nicholas K Taylor, Michael U Gutmann, Subramanian Ramamoorthy, and Kartic Subr. Adaptable pouring: Teaching robots not to spill using fast but approximate fluid simulation. In *Proceedings of the Conference on Robot Learning (CoRL)*, 2017.

[47] Xiaoxiao Guo, Satinder Singh, Honglak Lee, Richard L Lewis, and Xiaoshi Wang. Deep learning for real-time atari game play using offline monte-carlo tree search planning. In *International Conference on Neural Information Processing Systems (NIPS)*, pages 3338–3346, 2014.

[48] Joshua A Haustein, Jennifer King, Siddhartha S Srinivasa, and Tamim Asfour. Kinodynamic randomized rearrangement planning via dynamic transitions between statically stable states. In *Robotics and Automation (ICRA), 2015 IEEE International Conference on*, pages 3075–3082. IEEE, 2015.

[49] Mohammad Havaei, Axel Davy, David Warde-Farley, Antoine Biard, Aaron Courville, Yoshua Bengio, Chris Pal, Pierre-Marc Jodoin, and Hugo Larochelle. Brain tumor segmentation with deep neural networks. *Medical image analysis*, 35:18–31, 2017.

[50] Susan J Hespos, Alissa L Ferry, Erin M Anderson, Emily N Hollenbeck, and Lance J Rips. Five-month-old infants have general knowledge of how nonsolid substances behave and interact. *Psychological science*, 27(2):244–256, 2016.

[51] Susan J Hespos and Kristy VanMarle. Physics for infants: Characterizing the origins of knowledge about objects, substances, and number. *Wiley Interdisciplinary Reviews: Cognitive Science*, 3(1):19–27, 2012.

[52] Philip G Hill and Carl R Peterson. Mechanics and thermodynamics of propulsion. *Reading, MA, Addison-Wesley Publishing Co., 1992, 764 p.*, 1, 1992.

[53] Stefan Hinterstoisser, Vincent Lepetit, Slobodan Ilic, Stefan Holzer, Gary Bradski, Kurt Konolige, and Nassir Navab. Model based training, detection and pose estimation of texture-less 3d objects in heavily cluttered scenes. In *Asian conference on computer vision*, pages 548–562. Springer, 2012.

[54] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[55] Jamshed Iqbal, Zeashan Hameed Khan, and Azfar Khalid. Prospects of robotics in food industry. *Food Science and Technology*, 37(2):159–165, 2017.

[56] Scott C James, Yushan Zhang, and Fearghal O'Donncha. A machine learning framework to forecast wave conditions. *Coastal Engineering*, 137:1–10, 2018.

[57] Eric Jang, Sudheendra Vijaynarasimhan, Peter Pastor, Julian Ibarz, and Sergey Levine. End-to-end learning of semantic grasping. In *Conference on Robot Learning (CoRL)*, 2017.

[58] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 675–678, 2014.

[59] Wadim Kehl, Fabian Manhardt, Federico Tombari, Slobodan Ilic, and Nassir Navab. Ssd-6d: Making rgb-based 3d detection and 6d pose estimation great again. In *Proceedings of the International Conference on Computer Vision (ICCV 2017), Venice, Italy*, pages 22–29, 2017.

[60] Monroe Kennedy, Kendall Queen, Dinesh Thakur, Kostas Daniilidis, and Vijay Kumar. Precise dispensing of liquids using visual feedback. In *Proceedings of the International Conference on Intelligent Robots and Systems (IROS)*, pages 1260–1266, 2017.

[61] Jennifer E King, Joshua A Haustein, Siddhartha S Srinivasa, and Tamim Asfour. Nonprehensile whole arm rearrangement planning on physics manifolds. In *Robotics and Automation (ICRA), 2015 IEEE International Conference on*, pages 2508–2515. IEEE, 2015.

[62] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *Proceedings of the 3rd International Conference for Learning Representations*, San Diego, CA, USA, 2015.

[63] Reinhard Klapfer, Lars Kunze, and Michael Beetz. Pouring and mixing liquidsunderstanding the physical effects of everyday robot manipulation actions. *Human Reasoning and Automated Deduction*, 2012.

[64] Jonathan Ko, Daniel J Klein, Dieter Fox, and Dirk Haehnel. Gaussian processes and reinforcement learning for identification and control of an autonomous blimp. In

International Conference on Robotics and Automation (ICRA), pages 742–747. IEEE, 2007.

[65] N. Kohl and P. Stone. Policy gradient reinforcement learning for fast quadrupedal locomotion. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 2619–2624, 2004.

[66] GD Konidaris. *Autonomous Robot Skill Acquisition.* PhD thesis, University of Massachusetts, Amherst, 2011.

[67] George Konidaris and Andre S. Barreto. Skill discovery in continuous reinforcement learning domains using skill chaining. In *Advances in Neural Information Processing Systems 22*, pages 1015–1023, 2009.

[68] George Konidaris, Scott Kuindersma, Roderic Grupen, and Andre S. Barreto. Constructing skill trees for reinforcement learning agents from demonstration trajectories. In *Advances in Neural Information Processing Systems 23*, pages 1162–1170, 2010.

[69] Carolin Körner, Thomas Pohl, Ulrich Rüde, Nils Thürey, and Thomas Zeiser. Parallel lattice boltzmann methods for cfd applications. In *Numerical Solution of Partial Differential Equations on Parallel Computers*, pages 439–466. Springer, 2006.

[70] Michael C Koval, Jennifer E King, Nancy S Pollard, and Siddhartha S Srinivasa. Robust trajectory selection for rearrangement planning as a multi-armed bandit problem. In *IROS*, pages 2678–2685, 2015.

[71] Tibor Kozek and Tamás Roska. A double timescale cnn for solving two-dimensional navierstokes equations. *International Journal of Circuit Theory and Applications*, 24(1):49–55, 1996.

[72] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[73] Lars Kunze. *Naïve Physics and Commonsense Reasoning for Everyday Robot Manipulation.* PhD thesis, Technische Universität München, 2014.

[74] Lars Kunze and Michael Beetz. Envisioning the qualitative effects of robot manipulation actions using simulation-based projections. *Artificial Intelligence*, 2015.

[75] Yoshifumi Kuriyama, Ken'ichi Yano, and Masafumi Hamaguchi. Trajectory planning for meal assist robot considering spilling avoidance. In *Proceedings of the International Conference on Control Applications (CCA)*, pages 1220–1225, 2008.

[76] Jonathan P Kyle and Elon J Terrell. Application of smoothed particle hydrodynamics to full-film lubrication. *Journal of Tribology*, 135(4):041705, 2013.

[77] L'ubor Ladický, SoHyeon Jeong, Barbara Solenthaler, Marc Pollefeys, and Markus Gross. Data-driven fluid simulations using regression forests. *ACM Transactions on Graphics (TOG)*, 34(6):199:1–199:9, 2015.

[78] Matthew Lanaro, David P Forrestal, Stefan Scheurer, Damien J Slinger, Sam Liao, Sean K Powell, and Maria A Woodruff. 3d printing complex chocolate objects: Platform design, optimization and evaluation. *Journal of Food Engineering*, 215:13–22, 2017.

[79] Joshua D Langsfeld, Krishnanand N Kaipa, Rodolphe J Gentili, James A Reggia, and Satyandra K Gupta. Incorporating failure-to-success transitions in imitation learning for a dynamic pouring task. In *IEEE International Conference on Intelligent Robots and Systems (IROS) Workshop on Compliant Manipulation*, 2014.

[80] Steven M LaValle and James J Kuffner Jr. Randomized kinodynamic planning. *The international journal of robotics research*, 20(5):378–400, 2001.

[81] Vincent Lepetit, Pascal Fua, et al. Monocular model-based 3d tracking of rigid objects: A survey. *Foundations and Trends® in Computer Graphics and Vision*, 1(1):1–89, 2005.

[82] Sergey Levine, Chelsea Finn, Trevor Darrell, and Pieter Abbeel. End-to-end training of deep visuomotor policies. *The Journal of Machine Learning Research*, 17(1):1334–1373, 2016.

[83] Sergey Levine and Vladlen Koltun. Guided policy search. In *Proceedings of The 30th International Conference on Machine Learning*, pages 1–9, 2013.

[84] Yi Li, Gu Wang, Xiangyang Ji, Yu Xiang, and Dieter Fox. Deepim: Deep iterative matching for 6d pose estimation. In *European Conference Computer Vision (ECCV)*, 2018.

[85] Sai-Ho Ling, Herbert Ho-Ching Iu, Frank Hung-Fat Leung, and Kit Yan Chan. Improved hybrid particle swarm optimized wavelet neural network for modeling the development of fluid dispensing for electronic packaging. *IEEE transactions on industrial electronics*, 55(9):3447–3460, 2008.

[86] Song Liu, De Xu, You-Fu Li, Fei Shen, and Da-Peng Zhang. Nanoliter fluid dispensing based on microscopic vision and laser range sensor. *IEEE Transactions on Industrial Electronics*, 64(2):1292–1302, 2017.

[87] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. In *IEEE International Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 3431–3440, 2015.

[88] Yun Long, Xueyuan She, and Saibal Mukhopadhyay. Hybridnet: Integrating model-based and data-driven learning to predict evolution of dynamical systems. *arXiv preprint arXiv:1806.07439*, 2018.

[89] Adam Macdonald. *Fluidix*. OneZero Software, Canada, 2017.

[90] Miles Macklin and Matthias Müller. Position based fluids. *ACM Transactions on Graphics (TOG)*, 32(4):104:1–104:12, 2013.

[91] Miles Macklin, Matthias Müller, Nuttapong Chentanez, and Tae-Yong Kim. Unified particle physics for real-time applications. *ACM Transactions on Graphics (TOG)*, 33(4):104, 2014.

[92] Aleka McAdams, Eftychios Sifakis, and Joseph Teran. A parallel multigrid poisson solver for fluids simulation on large grids. In *Proceedings of the 2010 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 65–74. Eurographics Association, 2010.

[93] Amy McGovern and Andrew G Barto. Automatic discovery of subgoals in reinforcement learning using diverse density. In *Proceedings of the 18th International Conference on Machine Learning (ICML)*, 2001.

[94] Geoffrey McLachlan and David Peel. *Finite mixture models*. John Wiley & Sons, 2004.

[95] Heinrich Mellmann, Benjamin Schlotter, and Christian Blum. Simulation based selection of actions for a humanoid soccer-robot. In *Robot World Cup*, pages 193–205. Springer, 2016.

[96] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. In *Proceedings of the 27th International Conference on Neural Information Processing Systems (NIPS) Workshop on Deep Learning*, 2013.

[97] Igor Mordatch, Kendall Lowrey, and Emanuel Todorov. Ensemble-cio: Full-body dynamic motion planning that transfers to physical humanoids. In *Intelligent Robots and Systems (IROS), 2015 IEEE/RSJ International Conference on*, pages 5307–5314. IEEE, 2015.

[98] Toshio Morita, Hiroyasu Iwata, and Shigeki Sugano. Development of human symbiotic robot: Wendy. In *Robotics and Automation, 1999. Proceedings. 1999 IEEE International Conference on*, volume 4, pages 3183–3188. IEEE, 1999.

[99] Roozbeh Mottaghi, Hessam Bagherinezhad, Mohammad Rastegari, and Ali Farhadi. Newtonian image understanding: Unfolding the dynamics of objects in static images. In *Proceedings of the Conference of Computer Vision and Pattern Recognition (CVPR)*, 2016.

[100] Roozbeh Mottaghi, Mohammad Rastegari, Abhinav Gupta, and Ali Farhadi. "what happens if..." learning to predict the effect of forces in images. In *Proceedings of the European Conference on Computer Vision (ECCV)*, 2016.

[101] Roozbeh Mottaghi, Connor Schenck, Dieter Fox, and Ali Farhadi. See the glass half full: Reasoning about liquid containers, their volume and content. In *Proceedings of the International Conference on Computer Vision (ICCV)*, 2017.

[102] Damian Mrowca, Chengxu Zhuang, Elias Wang, Nick Haber, Li Fei-Fei, Joshua B Tenenbaum, and Daniel LK Yamins. Flexible neural representation for physics prediction. *arXiv preprint arXiv:1806.08047*, 2018.

[103] Matthias Müller, David Charypar, and Markus Gross. Particle-based fluid simulation for interactive applications. In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 154–159. Eurographics Association, 2003.

[104] Matthias Müller, Bruno Heidelberger, Marcus Hennix, and John Ratcliff. Position based dynamics. *Journal of Visual Communication and Image Representation*, 18(2):109–118, 2007.

[105] Bruce Roy Munson, Theodore Hisao Okiishi, Wade W Huebsch, and Alric P Rothmayer. *Fluid mechanics*. Wiley Singapore, 2013.

[106] S Niekum. *Semantically Grounded Learning from Unstructured Demonstrations*. PhD thesis, University of Massachusetts, Amherst, 2013.

[107] Scott Niekum. An integrated system for learning multi-step robotic tasks from unstructured demonstrations. In *AAAI Spring Symposium: Designing Intelligent Robots*, 2013.

[108] Kei Okada, Mitsuharu Kojima, Yuichi Sagawa, Toshiyuki Ichino, Kenji Sato, and Masayuki Inaba. Vision based behavior verification system of humanoid robot for daily environment tasks. In *IEEE-RAS International Conference on Humanoid Robotics (Humanoids)*, pages 7–12, 2006.

[109] Federico Pallottino, Liisa Hakola, Corrado Costa, Francesca Antonucci, Simone Figorilli, Anu Seisto, and Paolo Menesatti. Printing on food or food printing: a review. *Food and Bioprocess Technology*, 9(5):725–733, 2016.

[110] Yunpeng Pan, Ching-An Cheng, Kamil Saigol, Keuntaek Lee, Xinyan Yan, Evangelos Theodorou, and Byron Boots. Agile off-road autonomous driving using end-to-end deep imitation learning. In *Robotics: Science and Systems (RSS)*, 2018.

[111] Zherong Pan and Dinesh Manocha. Motion planning for fluid manipulation using simplified dynamics. In *Proceedings of the International Conference on Intelligent Robots and Systems (IROS)*, pages 4224–4231, 2016.

[112] Zherong Pan and Dinesh Manocha. Feedback motion planning for liquid pouring using supervised learning. In *Proceedings of the International Conference on Intelligent Robots and Systems (IROS)*, pages 1252–1259, 2017.

[113] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. In *In Proceedings of the Conference on Neural Information Processing Systems (NIPS) Workshop on Automatic Differentiation*, 2017.

[114] Vivian C Paulun, Takahiro Kawabe, Shinya Nishida, and Roland W Fleming. Seeing liquids from static snapshots. *Vision research*, 115:163–174, 2015.

[115] Georgios Pavlakos, Xiaowei Zhou, Aaron Chan, Konstantinos G Derpanis, and Kostas Daniilidis. 6-dof object pose from semantic keypoints. In *Robotics and Automation (ICRA), 2017 IEEE International Conference on*, pages 2011–2018. IEEE, 2017.

[116] Peter Pinggera, Toby Breckon, and Horst Bischof. On cross-spectral stereo matching using dense gradient features. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2012.

[117] Michael Posa, Cecilia Cantu, and Russ Tedrake. A direct method for trajectory optimization of rigid bodies through contact. *The International Journal of Robotics Research*, 33(1):69–81, 2014.

[118] Charles Ruizhongtai Qi, Li Yi, Hao Su, and Leonidas J Guibas. Pointnet++: Deep hierarchical feature learning on point sets in a metric space. In *Advances in Neural Information Processing Systems*, pages 5105–5114, 2017.

[119] Mahdi Rad and Vincent Lepetit. Bb8: A scalable, accurate, robust to partial occlusion method for predicting the 3d poses of challenging objects without using depth. In *International Conference on Computer Vision*, volume 1, page 5, 2017.

[120] Arturo Rankin and Larry Matthies. Daytime water detection based on color variation. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 215–221, 2010.

[121] Arturo L Rankin, Larry H Matthies, and Paolo Bellutta. Daytime water detection based on sky reflections. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 5329–5336, 2011.

[122] Junuthula Narasimha Reddy. *An Introduction to Nonlinear Finite Element Analysis: with applications to heat transfer, fluid mechanics, and solid mechanics.* OUP Oxford, 2014.

[123] Martin Riedmiller. Neural fitted Q iteration–First experiences with a data efficient neural reinforcement learning method. In *Proceedings of the 16th European Conference on Machine Learning (ECML)*, pages 317–328. Springer, 2005.

[124] Martin Riedmiller, Roland Hafner, Thomas Lampe, Michael Neunert, Jonas Degrave, Tom Van de Wiele, Volodymyr Mnih, Nicolas Heess, and Jost Tobias Springenberg. Learning by playing-solving sparse reward tasks from scratch. In *International Conference on Machine Learning (ICML)*, 2018.

[125] Bernardino Romera-Paredes and Philip Hilaire Sean Torr. Recurrent instance segmentation. In *European Conference on Computer Vision*, pages 312–329, 2016.

[126] Stephan Rosswog. Astrophysical smooth particle hydrodynamics. *New Astronomy Reviews*, 53(4-6):78–104, 2009.

[127] Leonel Rozo, Pedro Jimenez, and Carme Torras. Force-based robot learning of pouring skills using parametric hidden markov models. In *IEEE-RAS International Workshop on Robot Motion and Control (RoMoCo)*, pages 227–232, 2013.

[128] Radu Bogdan Rusu and Steve Cousins. 3d is here: Point cloud library (pcl). In *ICRA*, 2011.

[129] Connor Schenck and Dieter Fox. Guided policy search with delayed senor measurements. Technical report, University of Washington, Paul G. Allen School of Computer Science & Engineering, 2016.

[130] Connor Schenck and Dieter Fox. Towards learning to perceive and reason about liquids. In *Proceedings of the International Symposium on Experimental Robotics (ISER)*, 2016.

[131] Connor Schenck and Dieter Fox. Reasoning about liquids via closed-loop simulation. In *Robotics: Science & Systems (RSS)*, 2017.

[132] Connor Schenck and Dieter Fox. Visual closed-loop control for pouring liquids. In *Proceedings of the International Conference on Experimental Robotics (ICRA)*, 2017.

[133] Connor Schenck and Dieter Fox. Perceiving and reasoning about liquids using fully convolutional networks. *The International Journal of Robotics Research*, 37(4–5):452–471, 2018.

[134] Connor Schenck, Jonathan Tompson, Dieter Fox, and Sergey Levine. Learning robotic manipulation of granular media. In *Proceedings of the First Conference on Robotic Learning (CoRL)*, 2017.

[135] Tanner Schmidt, Katharina Hertkorn, Richard Newcombe, Zoltan Marton, Michael Suppa, and Dieter Fox. Depth-based tracking with physical constraints for robot manipulation. In *Robotics and Automation (ICRA), 2015 IEEE International Conference on*, pages 119–126. IEEE, 2015.

[136] Tanner Schmidt, Richard A Newcombe, and Dieter Fox. Dart: Dense articulated real-time tracking. In *Robotics: Science and Systems*, 2014.

[137] John Schulman, Alex Lee, Jonathan Ho, and Pieter Abbeel. Tracking deformable objects with point clouds. In *Robotics and Automation (ICRA), 2013 IEEE International Conference on*, pages 1130–1137. IEEE, 2013.

[138] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2016.

[139] Thierry Siméon, Jean-Paul Laumond, Juan Cortés, and Anis Sahbani. Manipulation planning with probabilistic roadmaps. *The International Journal of Robotics Research*, 23(7-8):729–746, 2004.

[140] Anand Pratap Singh, Shivaji Medida, and Karthik Duraisamy. Machine-learning-augmented predictive modeling of turbulent separated flows over airfoils. *AIAA Journal*, pages 2215–2227, 2017.

[141] William D Smart and Leslie Pack Kaelbling. Effective reinforcement learning for mobile robots. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 3404–3410, 2002.

[142] Barbara Solenthaler and Renato Pajarola. Predictive-corrective incompressible sph. In *ACM transactions on graphics (TOG)*, volume 28, page 40. ACM, 2009.

[143] Yu Sugimoto, Ken'ichi Yano, and Kazuhiko Terashima. Liquid level control of automatic pouring robot by two-degrees-of-freedom control. In *Proceedings of the 15th IFAC World Congress*, 2002.

[144] Minija Tamosiunaite, Bojan Nemec, Aleš Ude, and Florentin Wörgötter. Learning to pour with a robot arm combining goal and shape learning for dynamic movement primitives. *Robotics and Autonomous Systems*, 59(11):910–922, 2011.

[145] Yuval Tassa, Tom Erez, and Emanuel Todorov. Synthesis and stabilization of complex behaviors through online trajectory optimization. In *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*, pages 4906–4913. IEEE, 2012.

[146] Yuval Tassa, Nicolas Mansard, and Emo Todorov. Control-limited differential dynamic programming. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 1168–1175, 2014.

[147] Bugra Tekin, Sudipta N Sinha, and Pascal Fua. Real-time seamless single shot 6d object pose prediction. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 292–301, 2018.

[148] Roger Temam. *Navier-Stokes equations: theory and numerical analysis*, volume 343. American Mathematical Soc., 2001.

[149] Josh Tobin, Rachel Fong, Alex Ray, Jonas Schneider, Wojciech Zaremba, and Pieter Abbeel. Domain randomization for transferring deep neural networks from simulation to the real world. In *Intelligent Robots and Systems (IROS), 2017 IEEE/RSJ International Conference on*, pages 23–30. IEEE, 2017.

[150] Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*, pages 5026–5033. IEEE, 2012.

[151] Jonathan Tompson, Kristofer Schlachter, Pablo Sprechmann, and Ken Perlin. Accelerating eulerian fluid simulation with convolutional networks. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2017.

[152] Eric Tzeng, Coline Devin, Judy Hoffman, Chelsea Finn, Xingchao Peng, Sergey Levine, Kate Saenko, and Trevor Darrell. Towards adapting deep visuomotor representations from simulated to real environments. *arXiv preprint arXiv:1511.07111*, 2015.

[153] Abhinav Valada, Gabriel Oliveira, Thomas Brox, and Wolfram Burgard. Deep multispectral semantic scene understanding of forested environments using multimodal fusion. In *Proceedings of the International Symposium on Experimental Robotics (ISER)*, 2016.

[154] Kristy VanMarle and Karen Wynn. Tracking and quantifying objects and non-cohesive substances. *Developmental science*, 14(3):502–515, 2011.

[155] Damien Violeau. *Fluid Mechanics and the SPH method: theory and applications.* Oxford University Press, 2012.

[156] Doug Walker, Laura Smarandescu, and Brian Wansink. Half full or empty: Cues that lead wine drinkers to unintentionally overpour. *Substance Use & Misuse*, 49(3), 2014.

[157] Mike Wilson. Developments in robot applications for food manufacturing. *Industrial Robot: An International Journal*, 37(6):498–502, 2010.

[158] Jay M Wong, Vincent Kee, Tiffany Le, Syler Wagner, Gian-Luca Mariottini, Abraham Schneider, Lei Hamilton, Rahul Chipalkatty, Mitchell Hebert, David MS Johnson, et al. Segicp: Integrated deep semantic segmentation and pose estimation. In *Intelligent Robots and Systems (IROS), 2017 IEEE/RSJ International Conference on*, pages 5784–5789. IEEE, 2017.

[159] Yu Xiang and Dieter Fox. Da-rnn: Semantic mapping with data associated recurrent neural networks. In *Robots: Science & Systems*, 2017.

[160] Yu Xiang, Tanner Schmidt, Venkatraman Narayanan, and Dieter Fox. Posecnn: A convolutional neural network for 6d object pose estimation in cluttered scenes. *Robotics: Science and Systems (RSS)*, 2018.

[161] Zhe Xu and Maya Cakmak. Enhanced robotic cleaning with a low-cost tool attachment. In *Intelligent Robots and Systems (IROS 2014), 2014 IEEE/RSJ International Conference on*, pages 2595–2601. IEEE, 2014.

[162] Akihiko Yamaguchi and Christopher Atkeson. Differential dynamic programming for graph-structured dynamical systems: Generalization of pouring behavior with different skills. In *Proceedings of the International Conference on Humanoid Robotics (Humanoids)*, 2016.

[163] Akihiko Yamaguchi and Christopher Atkeson. Stereo vision of liquid and particle flow for robot pouring. In *Proceedings of the International Conference on Humanoid Robotics (Humanoids)*, 2016.

[164] Akihiko Yamaguchi and Christopher G Atkeson. Differential dynamic programming with temporally decomposed dynamics. In *IEEE-RAS International Conference on Humanoid Robotics (Humanoids)*, pages 696–703, 2015.

[165] Akihiko Yamaguchi and Christopher G Atkeson. Neural networks and differential dynamic programming for reinforcement learning problems. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 5434–5441, 2016.

[166] K Yano, T Toda, and K Terashima. Sloshing suppression control of automatic pouring robot by hybrid shape approach. In *Proceedings of the 40th IEEE Conference on Decision and Control (CDC)*, pages 1328–1333, 2001.

[167] Cha Zhang and Tsuhan Chen. Efficient feature extraction for 2d/3d objects in mesh representation. In *Image Processing, 2001. Proceedings. 2001 International Conference on*, volume 3, pages 935–938. IEEE, 2001.

[168] David Zheng, Vinson Luo, Jiajun Wu, and Joshua B Tenenbaum. Unsupervised learning of latent physical properties using perception-prediction networks. *arXiv preprint arXiv:1807.09244*, 2018.

[169] Wen-Hong Zhu and Joris De Schutter. Control of two industrial manipulators rigidly holding an egg. *IEEE control systems*, 19(2):24–30, 1999.