DEEP LEARNING FOR ROBOTICS

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Ian Lenz

January 2016

DEEP LEARNING FOR ROBOTICS

Ian Lenz, Ph.D.

Cornell University 2016

Robotics faces many unique challenges as robotic platforms move out of the lab and into the real world. In particular, the huge amount of variety encountered in real-world environments is extremely challenging for existing robotic control algorithms to handle. This necessistates the use of machine learning algorithms, which are able to learn controls given data. However, most conventional learning algorithms require hand-designed parameterized models and features, which are infeasible to design for many robotic tasks. Deep learning algorithms are general non-linear models which are able to learn features directly from data, making them an excellent choice for such robotics applications. However, care must be taken to design deep learning algorithms and supporting systems appropriate for the task at hand. In this work, I describe two applications of deep learning algorithms and one application of hardware neural networks to difficult robotics problems. The problems addressed are robotic grasping, food cutting, and aerial robot obstacle avoidance, but the algorithms presented are designed to be generalizable to related tasks.

BIOGRAPHICAL SKETCH

Ian Lenz was born February 22, 1988 in Pullman, WA. He first became interested in robotics through FIRST Robotics Team 639 at Ithaca High School in Ithaca, NY, acting as president for the 2006 season. He received a Bachelor of Science in Electrical and Computer Engineering with a minor in Robotics from Carnegie Mellon University in 2010.

ACKNOWLEDGEMENTS

I'd like to thank my committee – Ashutosh Saxena, Ross Knepper, Noah Snaveley, and Rajit Manohar – for all their advice and help. I'd also like to thank all my collaborators, in particular Honglak Lee and Mevlana Gemici, and my lab-mates, in particular Jaeyong Sung, Ashesh Jain, Yun Jiang, and Hema Koppula. A special thanks also to the huge team that made our aerial robot work possible, Dharmendra Modha, Shyamal Chandra, Thomas Zimmerman, and Myron Flickner at IBM, and Dale Cassidy, Jerry Yeh, Jasdeep Hundal, and Brian Wojcik at Cornell.

A huge thanks also to all my wonderful teachers through the years. In particular, I want to thank Rosely Teukolsky for being the greatest AP CS teacher any student could ask for. I can't overstate how much having such a strong foundation in programming and a wonderful teacher who encouraged interest has helped. Thanks also to all the advisors of the IHS Code Red Robotics team, including Scott Breigle, Mike Peters, Dave Buchner, Ian Krywe, Bill Brooks, Doug Fornell, and Roger Simpson. Code Red was my first taste of "real" robotics, and wouldn't have been possible without them.

On a more personal note, a huge thanks to my family, especially Mom and Eliot, who were a huge source of support throughout my PhD.

On a lighter note, I'd like to thank the folks that kept me fed through my PhD, especially the crews at Gorgers, the Red and White Cafe, and Mexeo, which is sorely missed. On an even ligher note, I'd like to thank Aaron Rodgers, since watching the Packers win has been more of a morale boost than it probably should've been.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

The field of robotics is at a very exciting point. Owing both to advances in algorithms and increasing computational power, robots are poised to move out of the lab and other special purpose applications such as manufacturing, and into our everyday lives. This can be seen especially from the recent upsurge in interest in autonomous vehicles, but robotics has the potential for even greater impact.

As robotics begins to move from the lab to the real world, robots face many new challenges. Consider a household personal assistant robot. Such a robot must perform many complex tasks, such as sorting and folding clothes, operating appliances, picking up and cleaning, and preparing food in the kitchen. Moreover, it must handle the huge variety of objects, materials, and the like associated with these tasks – for example, picking up different objects, some of which it may never have seen before, or preparing different food items. For many of these problems, there exists only an abstract relationship between the robot's visible inputs and the task at hand – for example, attempting to control from vision data, or determine material properties from haptic feedback.

Traditionally, a roboticist, or team thereof, would hand-design controllers for each task we want a robot to perform. Even for tasks which human users can perform intuitively, such as grasping objects or cutting food, these controllers can be very difficult to design because we aren't able to easily translate this natural intuition into code. It can also be extremely challenging to scale these approaches up to the huge amount of variety our robots must deal with in the real world – grasping every object in your home, cutting any food item, etc.

For these reasons, in recent years machine learning algorithms have seen widespread use for robotics applications. Rather than forcing the engineer to hand-code an entire end-to-end robotic system, machine learning allows portions of the system to be *learned* from some training data. This approach allows us to model concepts which might be difficult or impossible to properly hand-model. It also allows for adaptable models – as long as the form of the model is general, it can be adapted to more or different cases simply by providing training data for these new cases.

While machine learning algorithms have many advantages for robotic applications, they can still be difficult to apply to new problems. First, many learning algorithms require time-consuming optimization to perform inference, making them infeasible for robotic applications with strict time constraints. Designing models general enough to apply to other cases of the same problem, while still specific enough to be a good fit for the problem, can be very challenging, particularly with the huge variety seen in real-world robotics. In addition, most machine learning algorithms require significant hand-engineering in terms of designing *features* – transformations of the raw input, e.g. images, dynamics information, etc. given to the robot, into a form more useful for the learning algorithm. While designing good features is critical to the success of a machine learning algorithm for a particular problem, such features are often unitiutive, problem-specific, and take significant effort to design.

More recently, deep learning approaches have shown impressive performance across a wide range of domains, including computer vision, audio processing, natural language processing, and others. These algorithms are based on neural networks, highly-parameterized models which use multiple layers of representation to

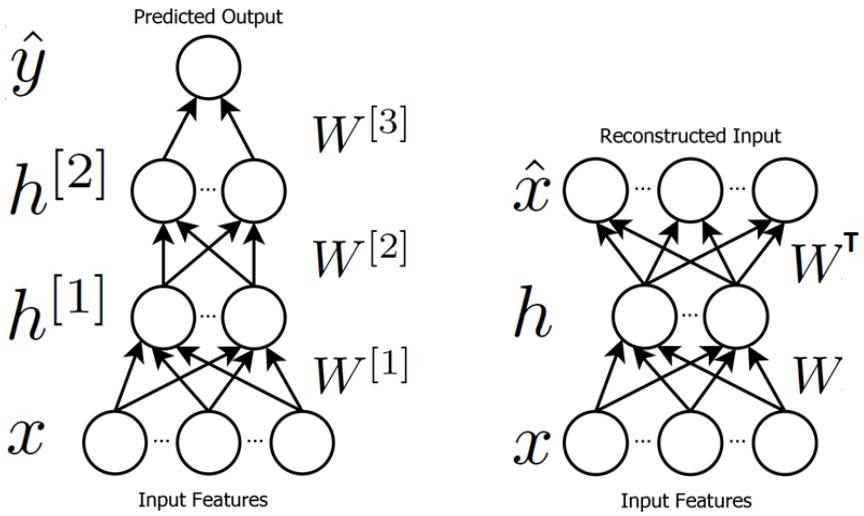


Figure 1.1: **Deep network and auto-encoder:** Left: A deep network with two hidden layers, which transform the input representation, and a logistic classifier at the top layer, which uses the features from the second hidden layer to predict output. Right: An auto-encoder, used for pretraining. A set of weights projects input features to a hidden layer. The same weights are then used to project these hidden unit outputs to a reconstruction of the inputs. In the sparse auto-encoder (SAE) algorithm, the hidden unit activations are also penalized.

transform data into a task-specific representation. By using unsupervised feature learning algorithms, deep learning approaches are able to pre-initialize these networks with useful features, avoiding the overfitting problems commonly seen when neural networks are trained without this initialization. This and several other useful properties, described in more detail below, make deep networks an excellent choice for robotic applications.

In this work, I will begin with a general description of deep learning algorithms and their strengths. I will then discuss their particular advantages as learning algorithms for robotics applications. Finally, I will present three applications of deep/neural network algorithms to diverse robotics problems – grasping, cutting unknown food materials, and aerial robot obstacle avoidance – highlighting the

strengths of these algorithms in real-world robotics applications.

1.1 Deep Learning

While deep learning is a broad class of learning algorithms, with many different associated inference models, most approaches learn a set of connection weights to be used by a neural network model for inference. A neural network, such as that shown in Fig. 1.1-left typically consists of multiple layers of artificial "neurons." Each neuron has weighted connections to each neuron in the previous layer, or to the network's inputs if the neuron sits at the lowest layer. The neuron then forms output by passing the weighted sum of its inputs through some nonlinear activation function (for example a sigmoid, $\sigma(a) = 1/(1 + exp(-a))$.) This output is then sent to the next layer. Ideally, each layer of features will represent a better abstraction of the input data, so that the final layer of features will be a better representation for some classifier than the raw features fed into the network.

For example, take $W^{[\ell]}$ as the network weights for layer ℓ, $h^{[\ell]}$ as the corresponding "hidden" representation generated using these weights, K_ℓ as the size of this hidden representation for layer l, x as the raw input features, N as the number of these features, and $\hat{y}$ as the network's predicted output, e.g. we want to model $P(\hat{y} = 1|x)$. Then, a simple two-layer deep network with a logistic classifier at the top layer might proceed as:

$$h_j^{[1]} = \sigma\left(\sum_{i=1}^{N} x_i W_{i,j}^{[1]}\right)$$
$$h_j^{[2]} = \sigma\left(\sum_{i=1}^{K_1} h_i^{[1]} W_{i,j}^{[2]}\right)$$

$$P(\hat{y} = 1|x; \Theta) = \sigma\left(\sum_{i=1}^{K_2} h_i^{[2]} W_i^{[3]}\right) \tag{1.1}$$

Note that, while the above network performs a binary classification task, it is simple to swap out the top-layer classification function for other tasks, e.g. using a softmax function for multi-class classification, or a linear weighting of the top-layer features for regression. This modularity is a major strength of neural network approaches.

Back-propagation: The above defined the inference procedure for a neural network, assuming the weights W had already been initialized. However, for this network to be useful, these weights must be learned from data to represent the nonlinearity we want to model. As with most learning algorithms, we have some cost function $C(P(\hat{y} = 1|x), y^*)$ which gives the cost of prediction probability $P(\hat{y} = 1|x)$ given ground-truth label y^*. During learning, we will optimize this cost over our entire dataset of training examples (x, y^*), e.g. taking $\Theta = \{W^{[1]}, W^{[2]}, W^{[3]}\}$ and using t to index M total training cases, and $C(X, Y, \Theta)$ as the cost across the entire dataset, learning proceeds as:

$$C(X, Y, \Theta) = \sum_{k=1}^{M} C(P(\hat{y}^{(t)} = 1|x^{(t)}), y^{*(t)}) \tag{1.2}$$

$$\Theta^* = \arg\min_{\Theta} C(X, Y, \Theta) \tag{1.3}$$

In most cases, we will use a gradient-based learning algorithm to optimize Θ. This requires the gradient of the cost function with respect to each parameter being optimized, i.e. $\partial C(X, Y, \Theta)/\partial\Theta$.

While gradients for the top-layer weights $W^{[3]}$ are the same as those for a

standard logistic classifier taking $h^{[2]}$ as input, gradients for the previous layers' weights are more complicated. For these, we use back-propagation, iteratively computing the gradient of the cost function with respect to the previous layer's hidden units, then using this gradient to compute the gradient with respect to that layer's weights, continuing this process until we have gradients for all these weights. Since the gradient of the cost function with respect to each layer's hidden units depends only on the same gradient for the next layer and the next layer's weights, this back-propagation requires only a single backwards pass through the network to compute all these gradients. Simplifying notation by defining $\partial C^{(t)}/\partial h^{[2](t)}$ as the gradient of the cost for case t with respect to $h^{[2]}$, we would back-propagate to the first-layer weights as:

$$\frac{\partial C^{(t)}}{\partial W_i^{[1]}} = \frac{\partial C^{(t)}}{\partial h^{[2](t)}} \frac{\partial h^{[2](t)}}{\partial h^{[1](t)}} \frac{\partial h^{[1](t)}}{\partial W_i^{[1]}} \tag{1.4}$$

This gives an efficient method for computing the gradients of the cost function with respect to each layer's weights. In Chapter 3, I will use a similar idea to efficiently compute cost-function gradients with respect to network *inputs*.

Unsupervised Feature Learning: Even moderately-sized neural networks will have a huge number of parameters which must be learned – for example, a network passing a 20x20 pixel image through two 200-unit layers will have 120,000 (20x20x200 + 200x200) different weights. This huge degree of parameterization has both advantages and disadvantages. While it allows these models to act as general nonlinear learners, capable of fitting nearly any function, it also makes them extremely vulnerable to overfitting, learning models which perform well on their training data but do not generalize to new cases. Such generalization is extremely important for robotics, which must operate in the real world, handling cases never seen before by the learning algorithm. This can be partly addressed

using regularization, or by increasing the amount of training data available. In Chapter 2 I will show that using specialized regularization, in particular, improves results. However, regularization is not a complete solution to overfitting, and can impact model performance if applied too strongly. Additional training data is more effective, but can be time-consuming to collect for complex robotic tasks.

Modern deep learning approaches use new learning algorithms to avoid these overfitting issues. Historical neural network methods would simply randomly initialize network weights, then back-propagate some cost function as described above. Since neural network optimization is inherently non-convex, it will converge only to a local minimum. Initializing the network randomly often leads this optimization to reach minima which overfit the training data. The core problem here is that, while each layer of the network is supposed to represent a more-useful abstraction of the input data, randomly initialized weights will not do so, forcing the algorithm to simultaneously learn a feature representation and a classifier from that initial poor feature representation.

Modern deep learning algorithms remedy this problem by using unsupervised feature learning algorithms. Rather than simultaneously learning a classifier and the features used for that classifier, these algorithms use unsupervised feature learning algorithms to initialize each layer of the network to a good representation. For example, we might intialize the previous network's first-layer weights using the sparse autoencoder (SAE) algorithm [46], as illustrated in Fig. 1.1-right, which optimizes a layer's weights to give a sparse representation which is able to reconstruct the layer's inputs. Taking $\hat{x}$ as the reconstruction of x, $g(a)$ as some sparsity function penalizing hidden-unit activations (e.g. the L1 sum of unit activations), and λ as a scaling factor defining the weighting between the two, we can

initialize $W^{[1]}$ as:

$$W^{[1]*} = \arg\min_{W^{[1]}} \sum_{t=1}^{M} (||\hat{x}^{(t)} - x^{(t)}||_2^2 + \lambda \sum_{j=1}^{K_1} g(h_j^{[1](t)})) \tag{1.5}$$

$$h_j^{[1](t)} = \sigma(\sum_{i=1}^{N} x_i^{(t)} W_{i,j}^{[1]})$$

$$\hat{x}_i^{(t)} = \sum_{j=1}^{K} h_j^{(t)} W_{i,j}^{[1]} \tag{1.6}$$

Since this algorithm is generic to its inputs, it could be re-used similarly to learn $W^{[2]}$ to give a sparse representation for $h^{[2]}$ which can reconstruct $h^{[1]}$. In this way, we can use the same feature learning algorithm to iteratively initialize our entire network, first learning lower layers, then fixing their outputs and learning features from them to learn the next layer's weights.

These feature learning approaches are one of the major strengths of modern deep learning methods. Since these algorithms are able to learn good features from data, they are much less sensitive to input representations than other conventional learning algorithms such as support vector machines, Gaussian processes, and others. Deep learning algorithms are able to learn good representations and solve problems even from basic representations such as raw pixels, avoiding the need to hand-design features as with other learning algorithms, saving significant engineering effort for many of the complex problems encountered in robotics, where features can be unintuitive and hard to design.

1.2 Deep Learning for Robotics

Robotics presents many unique challenges for learning algorithms. First, robots must perform a wide range of tasks, and it is often time-consuming or even infeasible to code completely new learning algorithms and features for each task. Second, robots must handle a huge amount of variety in the real world, which is difficult for many learning algorithms to handle. Finally, time is at a premium in most robotic applications, so learning algorithms must lend themselves to fast inference to be useful for robotic applications. Below, I will describe how the strengths of deep learning algorithms make them ideal choices for robotics.

Generality: Because deep networks are non-linear models with an extremely high number of parameters (typically on the order of millions), they are effectively general non-linear models, capable of learning any functional mapping from inputs to outputs. This is extremely useful for robotics applications, which typically encounter a huge range of nonlinearities, many of which are difficult or impossible to model. For example, both the mapping of pixels to graspability shown in Chapter 2 and the food-cutting dynamics shown in Chapter 3 would be extremely difficult to model by hand, but are able to be modeled by a deep network.

However, one significant caveat is that the huge number of learned parameters of a deep network also makes these models susceptible to high degrees of overfitting. This can be mitigated by designing network structures and learning algorithms which are a better fit for the form of the function being modeled. In Chapter 2, I will give a new learning algorithm which improves results for multimodal RGB-D data, and in Chapter 3 I will give a new recurrent deep architecture which learns latent features integrating long-term information for food-cutting and

other time-varying dynamics tasks. These two applications demonstrate the need for careful design of networks and learning algorithms, as using the right approach significantly improves results.

Feature Learning: As described above, modern deep learning techniques make use of unsupervised feature learning algorithms to learn good features from data to initialize the network. This allows the final back-propagation step to obtain better, more general results by starting from a good representation of the problem.

These feature learning methods are particularly important in robotics, as for many robotic tasks, it is very difficult to design useful features by hand. For example, hand-designing visual features useful for grasping, as discussed in Chapter 2 or features allowing us to model the complex dynamics involved in tasks like the food cutting discussed in Chapter 3 is extremely challenging. In Chapter 2, I will show that learned features can even outperform carefully hand-designed features.

Feature learning also aids genearlizability of these algorithms by adapting even the basic features used by the algorithm based on the given training data. This is useful because even applications using similar input data might require very different representations of that data. By contrast, using a traditional hand-engineering approach, we would either have to use the same features for both applications, which might weaken performance, or design new features for the second application, which would require significant effort. For example, adapting a system designed to cut food, as in Chapter 3, to another problem, such as scrubbing dishes, would require a very different set of features, but the deep learning algorithm I present in that chapter would be able to automatically learn these from data for that task.

Efficiency and Parallelism: Another major advantage of deep networks is their efficient, and natively parallel, inference. Fast inference is very important in robotics – a grasp detection algorithm which takes ten minutes to detect a grasp is not very useful in the real world. In more extreme cases, efficiency is a hard requirement – for example, real-time controllers often operate at rates of 100 or even 1000 Hz, and models used for model-predictive control (MPC), as in Chapter 3, must operate at similar rates to ensure the given controls are truly optimal.

In deep networks, inference typically consists of a series of matrix multiplications to weight inputs followed by element-wise non-linear operations (e.g. applying a sigmoid activation function.) Thus, inference does not require optimization, as is the case in other models such as conditional random fields (CRFs) and many others, allowing my model in Chapter 3 to predict at a rate of 1.2 kHz. Furthermore, such operations are extremely parallelizable – GPU implementations of deep network inference have become ubiquitious, significantly increasing performance. However, the structure of a deep network, composed of many individual "neurons," all peforming the same operation (except using different parameters), also lends itself well to direct parallel hardware implementations, which can be extremely efficient both in terms of time and power consumption, as shown in Chapter 4. The later, in particular, is critical for many battery-limited robotic platforms, such as miniature aerial vehicles.

Deep network inference time is also easily *scalable* – if faster inference (or a lower hardware footprint) is required, the number of units in the network can simply be reduced. While this might trade accuracy for performance, a similar tradeoff will likely be encountered when trying to speed up any machine learning algorithm.

However, most other learning algorithms would require significant work to similarly reduce the feature set, since the engineer would have to test different feature sets and weight the tradeoffs each gives in terms of accuracy vs. performance. Deep learning approaches let us simply define the size of the feature set to be learned and allow the algorithm to learn an optimal task-specific feature set of that size. I will demonstrate the power of this scalability in Chapter 3 as it allows that system to run at real-time rates with only a slight decrease in accuracy.

1.3 Related Work

In this section, I will describe some general related work in deep learning and robotic manipulation. In the following chapters, I will describe work related to each chapter's specific applications and methods.

1.3.1 Deep Learning

Modern deep learning methods retain the advantages of neural networks, while using new algorithms and network architectures to overcome their drawbacks. Due to their effectiveness as general non-linear learners [7], deep learning has been applied to a broad spectrum of problems, including visual recognition [50, 79], natural language processing [27], acoustic modeling [95], and many others. Recurrent deep networks have proven particularly effective for time-dependent tasks such as text generation [141] and speech recognition [47]. Factored conditional models using multiplicative interactions have also been shown to work well for modeling short-term temporal transformations in images [91]. More recently Taylor and Hinton

[143] applied these models to human motion, but did not model any control inputs, and treated the conditioning features as a set of fully-observed "motion styles". In Chapter 3, I will use both recurrent and factored conditional units to model the response of a dynamic system to control inputs.

Deep Learning for Manipulation

A few works have applied deep learning directly to robotic manipulation. Sung et al. [139] use deep learning to perform transfer learning for trajectories for manipulating household appliances. In both cases, their deep learning methods are limited to determining a manipulation plan – a grasping pose in the former case, and an end-effector trajectory in the latter – and then standard motion control algorithms are used to execute this plan. No deep networks are used for online control. Levine et al. [82] use a deep network to learn control policies. Their approach does not apply deep learning to modeling system dynamics, as I do in Chapter 3, and will be discussed in more detail in that chapter.

Deep Learning for Visual Detection: The majority of work in deep learning focuses on classification problems. Only a handful of previous works have applied these methods to detection problems [107, 78, 24]. For example, Osadchy et al. [107] and LeCun et al. [78] applied a deep energy-based model to the problem of face detection, Sermanet et al. [131] applied a convolutional neural network for pedestrian detection, and Coates et al. [24] used a deep learning approach to detect text in images. Girshick et al. [44] used learned convolutional features over image regions for object detection, while Szegedy et al. [142] used a multi-scale approach based on deep networks for the same task.

All these approaches focused on object detection and similar problems, in which

the goal is to find a bounding box which tightly contains the item to be detected, and for each item, all valid bounding boxes will be similar. In Chapter 2 I will apply a deep network to visual detection for robotic grasping with RGB-D data. In the grasp detection problem, there may be several valid grasps for an object in different regions, making it more important to select the one with the highest chance success. In addition, orientation matters much more to robotic grasp detection, as most grasps will only be viable for a small subset of the possible gripper orientations. My approach to grasp detection will also generalize across object classes, and even to classes never seen before by the system, as opposed to the class-specific nature of object detection.

Multimodal Deep Learning: Recent works in deep learning have extended these methods to handle multiple modalities of input data, such as audio and video [104], text and image data [138], and even RGB-D data [134, 14]. However, all of these approaches have fallen into two camps - either learning completely separate low-level features for each modality [104, 138], or simply concatenating the modalities [134, 14]. The former approaches have proven effective for data where the basic modalities differ significantly, such as the aforementioned case of text and images, while the latter is more effective in cases where the modalities are more similar, such as RGB-D data.

For some new combinations of modalities and tasks, it may not be clear which of these approaches will give better performance. In fact, in the ideal feature set, different features may use different subsets of the modalities. In Chapter 2, I will give a structured regularization method which guides the learning algorithm to select such subsets, without imposing hard constraints on network structure.

1.3.2 Robotic Manipulation

In this section, I will give a brief overview of related work in robotic manipulation problems similar to those presented in Chapters 2 and 3. I will present more detailed related work on those applications in those chapters.

Robotic Grasping

Robotic grasping has been an active research area for many years, but continues to be a challenging, unsolved problem. Many current works [32, 45, 148] synthesize grasps assuming a known 3D object model. While this allows for high-quality grasps, it does not allow these algorithms to generalize to new, unknown objects. My approach, presented in Chapter 2, and many others [128, 36, 54, 76, 61] use learning algorithms to detect grasps for novel objects from vision data. In contrast to these existing methods, my approach will *learn* even the basic features used to detect grasps, from raw pixel data. For a much more extensive overview of these and other robotic grasping methods, see Section 2.2.

Manipulating Deformable Objects

Many robotics works which manipulate deformable objects create task-specific systems and controllers. For example, Bollini et al. [16] developed such a system for baking cookies, Beetz et al. [6] for making pancakes, and Maitin-Shepard et al. [86] for towel folding. Gemici and Saxena [42] developed a more general system which learns to perform a range of tasks based on defined object properties. In Chapter 3, I will present a system which is general both to the task at hand and the specific objects and materials involved, learning a latent representation of these objects' properties. I refer the reader to Section 3.2 for a more detailed overview of these and other related works.

1.4 This Work

The remainder of this work will present three diverse applications of either deep learning or neural network methods to robotic tasks:

Chapter 2 – Deep Learning for Detecting Robotic Grasps: Here, the robot's goal is to grasp a *novel* object – i.e. one the robot has never seen before – using only a single frame of image and depth (RGB-D) data taken from a sensor such as a Microsoft Kinect. I treat this as a visual detection problem, and use a deep network to rank candidate grasps. This shows deep networks' ability to learn even abstract nonlinearities such as mapping RGB-D pixels to graspability. To improve detection efficiency, I employ a two-pass detection system where a smaller deep network is used to obtain a small set of top-ranked candidate grasps, which are then re-ranked by a larger network. I also present a new algorithm which uses structured regularization to learn more robust multimodal features, better integrating, for example, color and depth information obtained from Kinect. I validate these algorithms both in offline detection experiments on a large-scale grasping dataset and in real-world robotic experiments on both a PR2 and Baxter robot.

Chapter 3 – DeepMPC: Learning Deep Latent Features for Model Predictive Control: In this work, the robot's goal is to cut through an unknown food item at the fastest rate possible. This is an interesting control problem because of the complex, nonlinear dynamics (static friction, adhesion, deformation, etc.) and variety (different materials, varying temperatures, varying layers, etc.) involved in the problem. Due to this complexity, hand-coding effective controllers is extremely difficult, so instead I use a model-predictive controller (MPC) which

optimizes control inputs for some cost over predicted future system states. The chief difficulty in implementing an MPC algorithm, particularly for such a complex problem, lies in obtaining an accurate predictive dynamics model. To this end, I develop a new deep network and learning algorithm designed to handle the complex dynamics and variations present in food-cutting and other similar problems. I validate this model both in offline prediction experiments and using a real-time MPC system on a PR2 robot.

Chapter 4 – Low-Power Parallel Algorithms for Single Image based Obstacle Avoidance in Aerial Robots: Here, the goal is for an aerial robot to detect and avoid various types of obstacles from monocular vision. For aerial robots with limited battery life, power is at a premium, typically making processing hardware powerful enough to run e.g. vision algorithms. Thus, here, I use novel neural hardware with extremely low power consumption to implement the obstacle detection system. Since the initial obstacle maps inferred using only local classifiers are extremely noisy and would not be well-suited to use for real-world control, I implement belief propagation over a conditional random field in this neural hardware to integrate non-local information, significiantly improving results.

CHAPTER 2

DEEP LEARNING FOR DETECTING ROBOTIC GRASPS

2.1 Introduction

Robotic grasping is a challenging problem involving perception, planning, and control. Some recent works [124, 128, 61, 151] address the perception aspect of this problem by converting it into a detection problem in which, given a noisy, partial view of the object from a camera, the goal is to infer the top locations where a robotic gripper could be placed (see Figure 2.1). Unlike generic vision problems based on static images, such robotic perception problems are often used in closed loop with controllers, so there are stringent requirements on performance and computational speed. In the past, hand-designing features has been the most popular method for several robotic tasks [85, 71]. However, this is cumbersome and time-consuming, especially when incorporating new input modalities such as RGB-D cameras.

Recent methods based on deep learning [7] have demonstrated state-of-the-art performance in a wide variety of tasks, including visual recognition [75, 136], audio recognition [80, 95], and natural language processing [28]. These techniques are especially powerful because they are capable of learning useful features directly from both unlabeled and labeled data, avoiding the need for hand-engineering.

However, most work in deep learning has been applied in the context of *recognition*. Grasping is inherently a *detection* problem, and previous applications of

This work originally presented as a conference paper at Robotics: Science and Systems (RSS) 2013, and in the International Journal of Robotics Research (IJRR) Special Issue on Robot Vision 2015. This was joint work with Honglak Lee and Ashutosh Saxena.

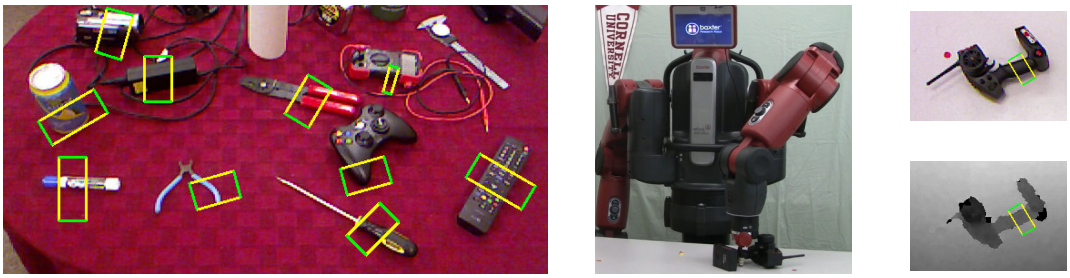


Figure 2.1: **Detecting robotic grasps:** Left: A cluttered lab scene labeled with rectangles corresponding to robotic grasps for objects in the scene. Green lines correspond to robotic gripper plates. I use a two-stage system based on deep learning to learn features and perform detection for robotic grasping. Center: A Baxter robot "Yogi" successfully executing a grasp detected by my algorithm. Right: The grasp detected for this case, in the RGB (top) and depth (bottom) images obtained from Kinect.

deep learning to detection have typically focused on specific vision applications such as face detection [107] and pedestrian detection [131]. The goal here is not only to infer a viable grasp, but to infer the optimal grasp for a given object that maximizes the chance of successfully grasping it. This differs significantly from the problem of object detection. Thus, the first major contribution of my work is to apply deep learning to the problem of robotic grasping, in a fashion which could generalize to similar detection problems.

The second major contribution of my work is to propose a new method for handling multimodal data in the context of feature learning. The use of RGB-D data, as opposed to simple 2D image data, has been shown to significantly improve grasp detection results [61, 32, 128]. In this work, I present a multimodal feature learning algorithm which adds a structured regularization penalty to the objective function to be optimized during learning. As opposed to previous works in deep learning, which either ignore modality information at the first layer (i.e., encourage all features to use all modalities) [134] or train separate first-layer features for each modality [104, 138], my approach allows for a middle-ground in which each feature is encouraged to use only a subset of the input modalities, but is not forced to use

only particular ones.

I also propose a two-stage cascaded detection system based on deep learning. Here, I use fewer features for the first pass, providing faster, but only approximately accurate detections. The second pass uses more features, giving more accurate detections. In my experiments, I found that the first deep network, with fewer features, was better at avoiding overfitting but less accurate. I feed the top-ranked rectangles from the first layer into the second layer, leading to robust early rejection of false positives. Unlike manually designed two-step features as in [61], my method uses deep learning, which allows me to learn detectors that not only give higher performance, but are also computationally efficient.

I test my approach on a challenging dataset, where I show that my algorithm improves both recognition and detection performance for grasping rectangle data. I also show that my two-stage approach is not only able to match the performance of a single-stage system, but, in fact, improves results while significantly reducing the computational time needed for detection.

In summary, the contributions of this work are:

- I present a deep learning algorithm for detecting robotic grasps. To the best of my knowledge, this is the first work to do so.
- In order to handle multimodal inputs, I present a new way to apply structured regularization to the weights to these inputs based on multimodal group regularization.
- I present a multi-step cascaded system for detection, significantly reducing its computational cost.
- My method outperforms the state-of-the-art for rectangle-based grasp detec-

tion, as well as previous deep learning algorithms.

- I implement my algorithm on both a Baxter and a PR2 robot, and show success rates of 84% and 89%, respectively, for executing grasps on a highly varied set of objects.

The rest of this work is organized as follows: I discuss related work in Section 2.2. I present my two-step cascaded detection system in Section 2.3, and some additional details in Section 2.4. I then describe my feature learning algorithm and structured regularization method in Section 2.5. I present my experiments in Section 2.6, and discuss results in Section 2.7. I then present experiments on both Baxter and PR2 robots in Section 2.8. I present several interesting directions for future work in Section 2.9, then conclude in Section 2.10.

2.2 Related Work

2.2.1 Robotic Grasping

In this section, I will focus on perception- and learning-based approaches for robotic grasping. For a more complete review of the field, I refer the reader to review papers by Bohg et al. [15], Sahbani et al. [122], Bicchi and Kumar [12] and Shimoga [133].

Most works define a "grasp" as an end-effector configuration which achieves partial or complete form- or force-closure of a given object. This is a challenging problem because it depends on the pose and configuration of the robotic gripper as well as the shape and physical properties of the object to be grasped, and typically requires a search over a large number of possible gripper configurations. Early

works [74, 105, 113] focused on testing for form- and force-closure, and synthesizing grasps fulfilling these properties according to some hand-designed "quality score" [39]. More recent works have refined these definitions [117]. These works assumed full knowledge of object shape and physical properties.

Grasping Given 3D Model: Fast synthesis of grasps for known 3D models remains an active research topic [32, 45, 148], with recent methods using advanced physical simulation to find optimal grasps. Gallegos et al. [41] performed optimization of grasps given both a 3D model of the object to be grasped and the desired contact points for the robotic gripper. Pokorny et al. [112] define spaces of graspable objects, then map new objects to these spaces to discover grasps. However, these works are only applicable when the full 3D model of the object is exactly known, which may not be the case when a robot is interacting with a new environment. I note that some of these physics-based approaches might be combined with my approach in a multi-pass system, discussed further in Sec. 2.9.

Sensing for Grasping: In a real-world robotic setting, a robot will not have full knowledge of the 3D model and pose of an object to be grasped, but rather only incomplete information from some set of sensors such as color or depth cameras, tactile sensors, etc. This makes the problem of grasping significantly more challenging [15], as the algorithm must use more limited and potentially noisier information to detect a good grasp. While some works [25, 108] simply attempt to estimate the poses of known objects and then apply full-model grasping algorithms based on these results, others avoid this assumption, functioning on novel objects which the algorithm has not seen before.

Such works often made use of other simplifying assumptions, such as assuming that objects belong to one of a set of primitive shapes [110, 17], or are planar

[98]. Other works produced impressive results for specific cases, such as grasping the corners of towels [85]. While such works escape the assumption of a fully-known object model, hand-coded grasping rules have a hard time dealing with the wide range of objects seen in real-world human environments, and are difficult and time-consuming to create.

Learning for Grasping: Machine learning methods have proven effective for a wide range of perception problems [146, 50, 79, 134, 14], allowing a perception system to learn a mapping from some feature set to various visual properties. Early work by Kamon et al. [65] showed that learning approaches could also be applied to the problem of grasping from vision, introducing a learning component to grasp quality scores.

Recent works have employed richer features and learning methods, allowing robots to grasp known objects which might be partially occluded [60] or in an unknown pose [31] as well as fully novel objects which the system has not seen before [124]. Here, I will address the latter case. Earlier work focused on detecting only a single grasping point from 2D partial-view data, using heuristic methods to determine a gripper pose based on this point. [126]. The use of 3D data was shown to significantly improve these results [128] thanks to giving direct physical information about the object in question. With the advent of low-cost RGB-D sensors such as the Kinect, the use of depth data for robotic grasping has become ubiquitous.

Several other works attempted to use the learning algorithm to more fully constrain the detected grasps. Ekvall and Kragic [36] and Huebner and Kragic [54] used shape-based approximations as bases for learning algorithms which directly gave an approach vector. Le et al. [76] treated grasp detection as a ranking problem

over sets of contact points in image space. Jiang et al. [61] represented a grasp as a 2D oriented rectangle in image space, with two edges corresponding to the gripper plates, using surface normals to determine the grasp approach vector. These approaches allow the detection algorithm to detect more exactly the gripper pose which should be used for grasping. In this work, I will follow the rectangle-based method.

Learning-based approaches have shown impressive results in grasping novel objects, showing that learning some parameters of the detection system can outperform human tuning. However, these approaches still require a significant degree of hand-engineering in the form of designing good input features.

Other Applications with RGBD Data. Due to the availability of inexpensive depth sensors, RGB-D data has been a significant research focus in recent years for various robotics applications. For example, Jiang et al. [63] consider robotic placement of objects, while Teuliere and Marchand [144] used RGB-D data for visual servoing. Several works, including those of Endres et al. [37] and Whelan et al. [149] have extended and improved Simultaneous Localization and Mapping (SLAM) for RGB-D data. Object detection and recognition has been a major focus in research on RGB-D data [26, 73, 20]. Most such works use hand-engineered features such as [121]. The few works that perform feature learning for RGB-D data [134, 14] largely ignore the multimodal nature of the data, not distinguishing the color and depth channels. Here, I present a structured regularization approach which allows me to learn more robust features for RGB-D and other multimodal data.

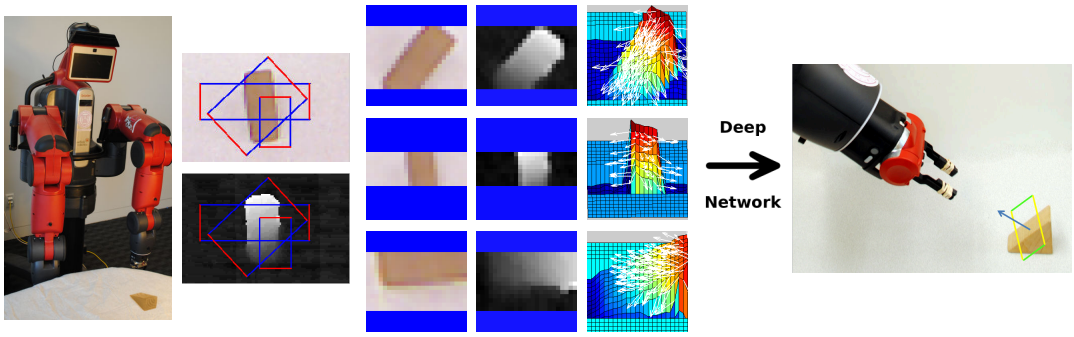


Figure 2.2: **Detecting and executing grasps:** From left to right: My system obtains an RGB-D image from a Kinect mounted on the robot, and searches over a large space of possible grasps, for which some candidates are shown. For each of these, it extracts a set of raw features corresponding to the color and depth images and surface normals, then uses these as inputs to a deep network which scores each rectangle. Finally, the top-ranked rectangle is selected and the corresponding grasp is executed using the parameters of the detected rectangle and the surface normal at its center. Red and green lines correspond to gripper plates, blue in RGB-D features indicates masked-out pixels.

2.2.2 Deep Learning

Structured Learning and Structured Regularization: Several approaches have been proposed which attempt to use a specially-designed regularization function to impose structure on a set of learned parameters without directly enforcing it. Jalali et al. [59] used a group regularization function in the multitask learning setting, where one set of features is used for multiple tasks. This function applies high-order regularization separately to particular groups of parameters. Their function regularized the number of features used for each task in a set of multi-class classification tasks solved by softmax regression. Intuitively, this encodes the belief that only some subset of the input features will be useful for each task, but this set of useful features might vary between tasks.

A few works have also explored the use of structured regularization in deep learning. The Topographic ICA algorithm [55] is a feature-learning approach that applies a similar penalty term to feature activations, but not to the weights them-

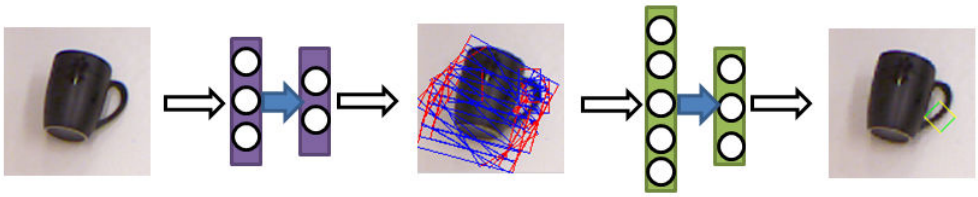


Figure 2.3: **Illustration of my two-stage detection process:** Given an image of an object to grasp, a small deep network is used to exhaustively search potential rectangles, producing a small set of top-ranked rectangles. A larger deep network is then used to find the top-ranked rectangle from these candidates, producing a single optimal grasp for the given object.

selves. Coates and Ng [23] investigate the problem of selecting receptive fields, i.e., subsets of the input features to be used together in a higher-level feature. The structure of the network is learned first, then fixed before learning the parameters of the network.

2.3 Deep Learning for Grasp Detection: System and Model

In this work, I will present an algorithm for robotic grasp detection from a single RGB-D view. My approach will be based on machine learning, but distinguish itself from previous approaches by learning not only the weights used to rank prospective grasps, but also the *features* used to rank them, which were previously hand-engineered.

I will do this using deep learning methods, learning a set of RGB-D features which will be extracted from each candidate grasp, then used to score that grasp. My approach will include a structured multimodal regularization method which improves the quality of the features learned from RGB-D data without constraining

network structure.

In my system for robotic grasping, as shown in Fig. 2.2, the robot first obtains an RGB-D image of the scene containing objects to be grasped. A small deep network is used to score potential grasps in this image, and a small candidate set of the top-ranked grasps is provided to a larger deep network, which yields a single best-ranked grasp.

In this work, I will represent potential grasps using oriented rectangles in the image plane as seen on the left in Fig. 2.2, with one pair of parallel edges corresponding to the robotic gripper [61]. Each rectangle is thus parameterized by the X and Y coordinates of its upper-left corner, its width, height, and orientation in the image plane, giving a five-dimensional search space for potential grasps. Grasps will be ranked based on features extracted from the RGB-D image region contained inside their corresponding rectangle, aligned to the gripper plates, as seen in the center of Fig. 2.2.

To translate a rectangle such as that shown on the right in Fig. 2.2 into a gripper pose for grasping I find the point with the minimum depth inside the central third (horizontally) of the rectangle. I then use the averaged surface normal around this point to determine the approach vector for the gripper. The orientation of the detected rectangle is translated to a rotation around this vector to orient the gripper. I use the X-Y coordinates of the rectangle center along with the depth of the closest point to determine a grasping point in the robot's coordinate frame. I compute a pre-grasp position by shifting 10 cm back from the grasping point along this approach vector and position the gripper at this point. I then approach the object along the approach vector and grasp it.

Using a standard feature learning approach such as sparse auto-encoder [46], a deep network can be trained for the problem of grasping rectangle recognition (i.e., does a given rectangle in image space correspond to a valid robotic grasp?). However, in a real-world robotic setting, my system needs to perform *detection* (i.e., given an image containing an object, how should the robot grasp it?). This task is significantly more challenging than simple recognition.

Two-stage Cascaded Detection: In order to perform detection, one naive approach could be to consider each possible oriented rectangle in the image (perhaps discretized to some level), and evaluate each rectangle with a deep network trained for recognition. However, such near-exhaustive search of possible rectangles (based on positions, sizes, and orientations) can be quite expensive in practice for real-time robotic grasping.

Motivated by multi-step cascaded approaches in previous work [61, 146], I instead take a two-stage approach to detection: First, I use a reduced feature set to determine a set of top candidates. Then, I use a larger, more robust feature set to rank these candidates.

However, these approaches require the design of two separate sets of features. In particular, it can be difficult to manually design a small set of first-stage features which is both quick to compute and robust enough to produce a good set of candidate detections for the second stage. Using deep learning allows me to circumvent the costly manual design of features by simply training networks of two different sizes, using the smaller for the exhaustive first pass, and the larger to re-rank the candidate detection results.

Model: To detect robotic grasps from the rectangle representation, I model the

probability of a rectangle $G^{(t)}$, with features $x^{(t)} \in \mathbb{R}^N$ being graspable, using a random variable $\hat{y}^{(t)} \in \{0, 1\}$ which indicates whether or not I predict $G^{(t)}$ to be graspable. I use a deep network similar to that previously defined in Section 1.1 to model the probability of this rectangle being graspable, $P(\hat{y}^{(t)} = 1|x^{(t)})$.

2.3.1 Inference and Learning

During **inference**, my goal is to find the single grasping rectangle with the maximum probability of being graspable for some new object. With G representing a particular grasping rectangle position, orientation, and size, I find this best rectangle as:

$$G^* = \arg\max_G P(\hat{y}^{(t)} = 1|\phi(G); \Theta) \tag{2.1}$$

Here, the function ϕ extracts the appropriate input representation for rectangle G.

During **learning**, my goal is to learn the parameters Θ that optimize the recognition accuracy of my system. Here, input data is given as a set of pairs of features $x^{(t)} \in \mathbb{R}^N$ and ground-truth labels $y^{(t)} \in \{0, 1\}$ for $t = 1, \ldots, M$. As described above I use a two-phase learning approach. First, I pre-train the hidden-layer weights $W^{[1]}$ and $W^{[2]}$ using an algorithm similar to the SAE approach given in Equation 1.6, but also including a regularization term, $f(W)$, weighted by β:

$$W^* = \arg\min_W \sum_{t=1}^{M} (||\hat{x}^{(t)} - x^{(t)}||_2^2 + \lambda \sum_{j=1}^{K} g(h_j^{(t)})) + \beta f(W) \tag{2.2}$$

During the *supervised* phase of the learning algorithm, I then jointly learn classifier weights $W^{[3]}$ and fine-tune hidden layer weights $W^{[1]}$ and $W^{[2]}$ for recognition,

using back-propagation as described in Section 1.1. I maximize the log-likelihood of the data along with regularization penalties on hidden layer weights:

$$\Theta^* = \arg\max_{\Theta} \sum_{t=1}^{M} \log P(\hat{y}^{(t)} = y^{(t)} | x^{(t)}; \Theta) - \beta_1 f(W^{[1]}) - \beta_2 f(W^{[2]}) \quad (2.3)$$

Two-stage Detection Model: During **inference** for two-stage detection, I will first use a smaller network to produce a set of the top T rectangles with the highest probability of being graspable according to network parameters Θ_1. I will then use a larger network with a separate set of parameters Θ_2 to re-rank these T rectangles and obtain a single best one. The only change to **learning** for the two-stage model is that these two sets of parameters are learned separately, using the same approach.

2.4 System Details

In this section, I will define the set of raw features which my system will use, forming x in the equations above, and how they are extracted from an RGB-D image. Some examples of these features are shown in Fig 2.2.

My algorithm uses only local information - specifically, I extract the RGB-D sub-image contained within each rectangle, and use this to generate features for that rectangle. This image is rotated so that its left and right edges correspond to the gripper plates, and then re-scaled to fit inside the network's receptive field.

From this 24x24 pixel image, seven channels' worth of features are extracted,

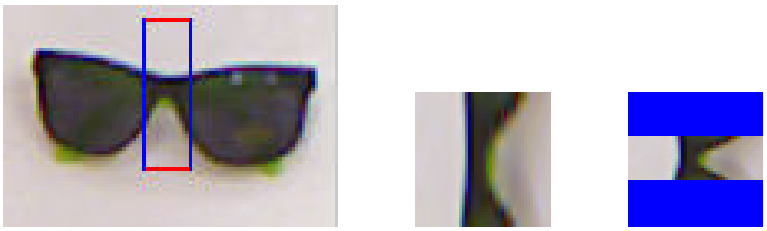


Figure 2.4: **Preserving aspect ratio:** Left: a pair of sunglasses with a potential grasping rectangle. Red edges indicate gripper plates. Center: image taken from the rectangle and rescaled to fit a square aspect ratio. Right: same image, padded and centered in the receptive field. Blue areas indicate masked-out padding. When rescaled, the rectangle incorrectly appears graspable. Preserving aspect ratio and padding allows the rectangle to correctly appear non-graspable.

Figure 2.5: **Improvement from mask-based scaling:** Left: Result without mask-based scaling. Right: Result with mask-based scaling.

giving 24x24x7 = 4032 input features. The first three channels are the image in YUV color space, used because it represents image intensity and color separately. The next is simply the depth channel of the image. The last three are the X, Y, and Z components of surface normals computed based on the depth channel. These are computed after the image is aligned to the gripper so that they are always relative to the gripper plates.

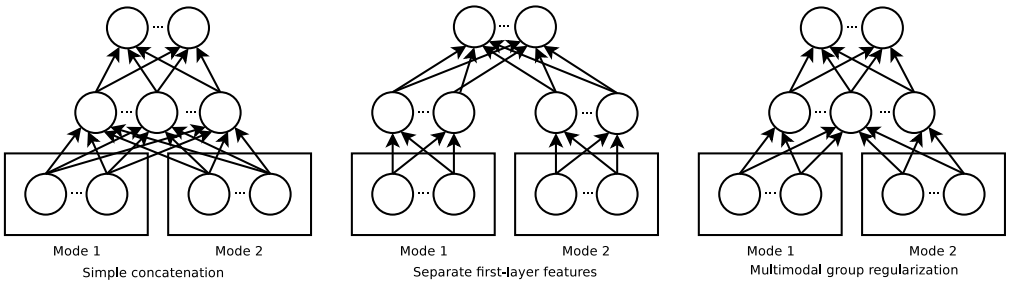


Figure 2.6: **Three possible models for multimodal deep learning:** Left: fully dense model—all visible features are concatenated and modality information is ignored. Middle: modality-specific sparse model - separate first layer features are trained for each modality. Right: group-sparse model—a structured regularization term encourages features to use only a subset of the input modes.

2.4.1 Data Pre-Processing

Whitening data is critical for deep learning approaches to work well, especially in cases such as multimodal data where the statistics of the input data may vary greatly. While PCA-based approaches have been shown to be effective [56], they are difficult to apply in cases such as mine where large portions of the data may be masked out.

Depth data, in particular, can be difficult to whiten because the range of values may be very different for different patches in the image. Thus, I first whiten each depth patch individually, subtracting the patch-wise mean and dividing by the patch-wise standard deviation, down to some minimum.

For multimodal data, the statistics of the data for each modality should match as closely as possible, to avoid learning features which are biased towards or away from using particular modes. This is particularly important when regularizing each modality separately, as in my approach. Thus, I drop mean values for each feature separately, but scale the data for each channel by dividing by the standard deviation of all its features combined.

2.4.2 Preserving Aspect Ratio

It is important for to preserve aspect ratio when feeding features into the network. This is because distorting image features may cause non-graspable rectangles to appear graspable, as shown in Fig. 2.4. However, padding with zeros can cause rectangles with less padding to receive higher graspability scores, as the network will have more nonzero inputs. It is important to account for this because in many cases the ideal grasp for an object might be represented by a thin rectangle which would thus contain many zero values in its receptive field from padding.

To address this problem, I scale up the magnitude of the available input for each rectangle based on the fraction of the rectangle which is masked out. In particular, I define a multiplicative scaling factor for the inputs from each modality, based on the fraction of each mode which is masked out, since each mode may have a different mask.

In the multimodal setting, I assume that the input data x is known to come from R distinct modalities, for example audio and video data, or depth and RGB data. I define the modality matrix S as an RxN binary matrix, where each element $S_{r,i}$ indicates membership of visible unit x_i in a particular modality r, such as depth or image intensity. The scaling factor for mode r is then defined as: $\Psi_r^{(t)} = \sum_{i=1}^{N} S_{r,i} / \left(\sum_{i=1}^{N} S_{r,i} \mu_i^{(t)} \right)$, where $\mu_i^{(t)}$ is 1 if $x_i^{(t)}$ is masked in, 0 otherwise. The scaling factor for case i is: $\psi_i^{(t)} = \sum_{r=1}^{R} S_{r,i} \Psi_r^{(t)}$.

I could simply scale up each value of x by its corresponding scale factor when training my model, as $x'^{(t)}_i = \psi_i^{(t)} x_i^{(t)}$. However, since my sparse autoencoder penalizes squared error, scaling x linearly will scale the error for the corresponding cases quadratically, causing the learning algorithm to lend increased significance

to cases where more data is masked out. Instead, I can use the scaled x' as input to the network, but penalize reconstruction based on the original x, only scaling after the squared error has been computed:

$$W^* = \arg\min_W \sum_{t=1}^{M} \left(\sum_{i=1}^{N} \psi_i^{(t)} (\hat{x}_i^{(t)} - x_i^{(t)})^2 + \lambda \sum_{j=1}^{K} g(h_j^{(t)}) \right) \tag{2.4}$$

I redefine the hidden units to use the scaled visible input:

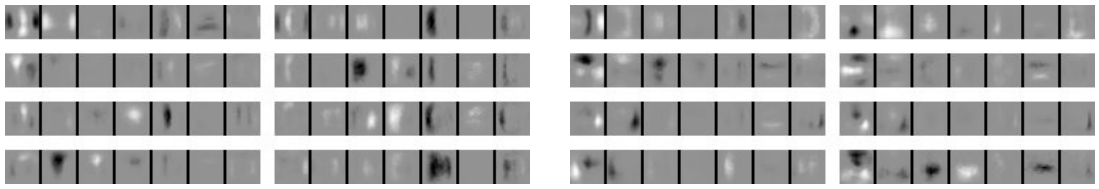
$$h_j^{(t)} = \sigma \left(\sum_{i=1}^{N} x'^{(t)}_i W_{i,j} \right) \tag{2.5}$$

This approach is equivalent to adding additional, potentially fractional, 'virtual' visible units to the model based on the scaling factor for each mode. In practice, I found it necessary to limit the scaling factor to a maximum of some value c, as $\Psi'^{(t)}_r = \min(\Psi_r^{(t)}, c)$.

As shown in Table 2.3 my mask-based scaling technique at the visible layer improves grasping results by over 25% for both metrics. As seen in Figure 2.5, it removes the network's inherent bias towards square rectangles, exhibiting a much wider range of aspect ratios that more closely matches that of the ground-truth data.

2.5 Structured Regularization for Feature Learning

A naive way of applying feature learning to multimodal data is to simply take x (as a concatenated vector) as input to the model described above, ignoring information about specific modalities, as seen on the lefthand side of Figure 2.6. This approach may either 1) prematurely learn features which include all modalities, which can lead to overfitting, or 2) fail to learn associations between modalities with very different underlying statistics.



(a) Features corresponding to positive grasps. (b) Features corresponding to negative grasps.

Figure 2.7: **Features learned from grasping data:** Each feature contains seven channels - from left to right, depth, Y, U, and V image channels, and X, Y, and Z surface normal components. Vertical edges correspond to gripper plates. Left: eight features with the strong positive correlations to rectangle graspability. Right: similar, but negative correlations. Group regularization eliminates many modalities from many of these features, making them more robust.

Instead of concatenating multimodal input as a vector, Ngiam et al. [104] proposed training a first layer representation for each modality separately, as shown in Figure 2.6-middle. This approach makes the assumption that the ideal low-level features for each modality are purely unimodal, while higher-layer features are purely multimodal. This approach may work better for some problems where the modalities have very different basic representations, such as the video and audio data (as used in [104]), so that separate first layer features may give better performance. However, for modalities such as RGB-D data, where the input modes represent different channels of an image, learning low-level correlations can lead to more robust features – my experiments in Section 2.6 show that simply concatenating the input modalities significantly outperforms training separate first-layer features for robotic grasp detection from RGB-D data.

For many problems, it may be difficult to tell which of these approaches will perform better, and time-consuming to tune and comparatively evaluate multiple algorithms. In addition, the ideal feature set for some problems may contain features which use some, but not all, of the input modalities, a case which neither of these approaches are designed to handle.

To solve these problems, I propose a new algorithm for feature learning for multimodal data. My approach incorporates a structured penalty term into the optimization problem to be solved during learning. This technique allows the model to learn correlated features between multiple input modalities, but regularizes the number of modalities used per feature (hidden unit), discouraging the model from learning weak correlations between modalities. With this regularization term, the algorithm can specify how mode-sparse or mode-dense the features should be, representing a continuum between the two extremes outlined above.

Regularization in Deep Learning: In a typical deep learning model, L_2 regularization (i.e., $f(W) = ||W||_2^2$) or L_1 regularization (i.e., $f(W) = ||W||_1$) are commonly used in training (e.g., as specified in Equations (2.2) and (2.3)). These are often called a "weight cost" (or "weight decay"), and are left implicit in many works.

Applying regularization is well known to improve the generalization performance of feature learning algorithms. One might expect that a simple L_1 penalty would eliminate weak correlations in multimodal features, leading to features which use only a subset of the modes each. However, I found that in practice, a value of β large enough to cause this also degraded the quality of features for the remaining modes and lead to decreased task performance.

Multimodal Regularization: Structured regularization, such as in [59], takes a set of groups of weights, and applies some regularization function (typically high-order) *separately* to each group. In my structured multimodal regularization algorithm, each modality will be used as a regularization group separately for each

hidden unit. For example, a group-wise p-norm would be applied as:

$$f(W) = \sum_{j=1}^{K} \sum_{r=1}^{R} \left(\sum_{i=1}^{N} S_{r,i} |W_{i,j}^{p}| \right)^{1/p} \tag{2.6}$$

where $S_{r,i}$ is 1 if feature i belongs to group r and 0 otherwise. Using a high value of p allows me to penalize higher-valued weights from each mode to each feature more strongly than lower-valued ones. This also means that forming a high-valued weight in a group with other high-valued weights will accrue a lower additional penalty than doing so for a group with only low-valued weights. At the limit ($p \to \infty$), this group regularization becomes equivalent to the infinity (or max) norm:

$$f(W) = \sum_{j=1}^{K} \sum_{r=1}^{R} \max_{i} S_{r,i} |W_{i,j}| \tag{2.7}$$

which penalizes only the maximum weight from each mode to each feature. In practice, the infinity norm is not differentiable and therefore is difficult to apply gradient-based optimization methods; here, I use the log-sum-exponential as a differentiable approximation to the max norm.

In experiments, this regularization function produces first-layer weights concentrated in fewer modes per feature. However, I found that at values of β sufficient to induce the desired mode-wise sparsity patterns, penalizing the maximum also had the undesirable side-effect of causing many of the weights for other modes to saturate at their mode's maximum, suggesting that the features were overly constrained. In some cases, constraining the weights in this manner also caused the algorithm to learn duplicate (or redundant) features, in effect scaling up the feature's contribution to reconstruction to compensate for its constrained maximum. This is obviously an undesirable effect, as it reduces the effective size (or diversity) of the learned feature set.

This suggests that the max-norm may be overly constraining. A more desirable sparsity function would penalize nonzero weight maxima for each mode for each feature without additional penalty for larger values of these maxima. I can achieve this effect by applying the L_0 norm, which takes a value of 0 for an input of 0, and 1 otherwise, on top of the max-norm from above:

$$f(W) = \sum_{j=1}^{K} \sum_{r=1}^{R} \mathbb{I}\{(\max_{i} S_{r,i}|W_{i,j}|) > 0\} \tag{2.8}$$

where $\mathbb{I}$ is the indicator function, which takes a value of 1 if its argument is true, 0 otherwise. Again, for a gradient-based method, I used an approximation to the L_0 norm, such as $\log(1 + x^2)$. This regularization function now encodes a direct penalty on the number of modes used for each weight, without further constraining the weights of modes with nonzero maxima.

Figure 2.7 shows features learned from the unsupervised stage of my group-regularized deep learning algorithm. I discuss these features, and their implications for robotic grasping, in Section 2.7.

2.6 Experiments

2.6.1 Dataset

I used the extended version of the Cornell grasping dataset for my experiments. This dataset, along with code for this chapter, is available at `http://pr.cs.cornell.edu/deepgrasping`. I note that this is an updated version of the dataset used in [61], containing several more complex objects, and thus results for their algorithms will be different from those in [61]. This dataset contains 1035

Figure 2.8: **Example objects from the Cornell grasping dataset:** [61]. This dataset contains objects from a large variety of categories.

images of 280 graspable objects, several of which are shown in Fig. 2.8. Each image is annotated with several ground-truth positive and negative grasping rectangles. While the vast majority of possible rectangles for most objects will be non-graspable, the dataset contains roughly equal numbers of graspable and non-graspable rectangles. I will show that this is useful for an unsupervised learning algorithm, as it allows learning a good representation for graspable rectangles even from unlabeled data.

I performed five-fold cross-validation, and present results for splits on per image (i.e., the training set and the validation set do not share the same image) and per object (i.e., the training set and the validation set do not share any images from the same object) basis. Hyper-parameters were selected by validating performance on a separate set of 300 grasps not used in any of the cross-validation splits.

I take seven 24x24 pixel channels as described in Section 2.4 as input, giving 4032 input features to each network. I trained a deep network with 200 hidden

units each at the first and second layers using my learning algorithm as described in Sections 2.3 and 2.5. Training this network took roughly 30 minutes. For trials involving my two-pass system, I trained a second network with 50 hidden units at each layer in the same manner. During inference I performed an exhaustive search using this network, then used the 200-unit network to re-rank the 100 highest-ranked rectangles found by the 50-unit network.

2.6.2 Baselines

I compare my recognition results in the Cornell grasping dataset with the features from [61], as well as the combination of these features and Fast Point Feature Histogram (FPFH) features [120]. I used a linear SVM for classification, which gave the best results among all other kernels. I also report chance performance, obtained by randomly selecting a label in the recognition case, and randomly assigning scores to rectangles in the detection case.

I also compare my algorithm to other deep learning approaches. I compare to a network trained only with standard L1 regularization, and a network trained in a manner similar to [104], where three separate sets of first layer features are learned for the depth channel, the combination of the Y, U, and V channels, and the combination of the X, Y, and Z surface normal components.

2.6.3 Metrics for Detection

For detection, I compare the top-ranked rectangle for each method with the set of ground-truth rectangles for each image. I present results using two metrics, the

Table 2.1: **Recognition results for Cornell grasping dataset.**

Algorithm	Accuracy (%)
Chance	50
Jiang et al. [61]	84.7
Jiang et al. [61] + FPFH	89.6
Sparse AE, separate layer-1 feat.	92.8
Sparse AE	**93.7**
Sparse AE, group reg.	**93.7**

"point" and "rectangle" metric.

For the point metric, similar to Saxena et al. [126], I compute the center point of the predicted rectangle, and consider the grasp a success if it is within some distance from at least one ground-truth rectangle center. I note that this metric ignores grasp orientation, and therefore might overestimate the performance of an algorithm for robotic applications.

For the rectangle metric, similar to Jiang et al. [61], let G be the top-ranked grasping rectangle predicted by the algorithm, and G^* be a ground-truth rectangle. Any rectangles with an orientation error of more than 30^o from G are rejected. From the remaining set, I use the common bounding box evaluation metric of intersection divided by union - i.e. $Area(G \cap G^*)/Area(G \cup G^*)$. Since a ground-truth rectangle can define a large space of graspable rectangles (e.g., covering the entire length of a pen), I consider a prediction to be correct if it scores at least 25% by this metric.

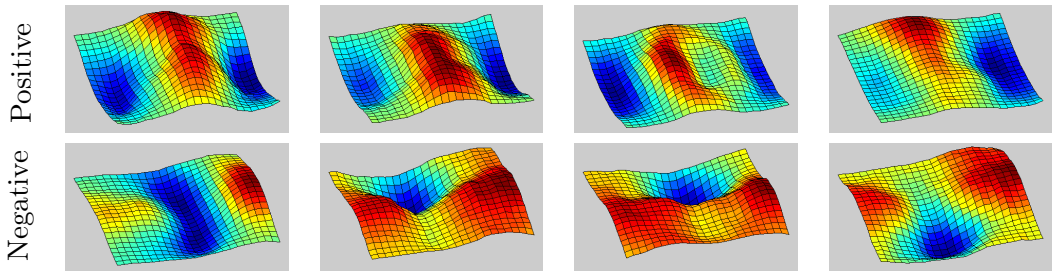


Figure 2.9: **Learned 3D depth features:** 3D meshes for depth channels of the four features with strongest positive (top) and negative(bottom) correlations to rectangle graspability. Here X and Y coordinates corresponds to positions in the deep network's receptive field, and Z coordinates corresponds to weight values to the depth channel for each location. Feature shapes clearly correspond to graspable and non-graspable structures, respectively.

2.7 Results and Discussion

2.7.1 Deep Learning for Robotic Grasp Detection

Figure 2.7 shows the features learned by the unsupervised phase of my algorithm which have a high correlation to positive and negative grasping cases. Many of these features show non-zero weights to the depth channel, indicating that it learns the correlation of depths to graspability. I can see that weights to many of the modalities for these features have been eliminated by my structured regularization approach. In particular, many of these features lack weights to the U and V (3^{rd} and 4^{th}) channels, which correspond to color, allowing the system to be more robust to different-colored objects.

Figure 2.9 shows 3D meshes for the depth channels of the four features with the strongest positive and negative correlations to valid grasps. Even *without any supervised information*, my algorithm was able to learn several features which correlate strongly to graspable cases and non-graspable cases. The first two positive-correlated features represent handles, or other cases with a raised region in the

Table 2.2: **Recognition results for different modalities,** for a deep network pre-trained using SAE.

Modes	Accuracy (%)
Chance	50
RGB	90.3
Depth	92.4
Surf. Normals	90.3
Depth + Surf. Normals	92.8
RGB + Depth + Surf. Normals	**93.7**

center, while the second two represent circular rims or handles. The negatively-correlated features represent obviously non-graspable cases, such as ridges perpendicular to the gripper plane and "valleys" between the gripper plates. From these features, I can see that even during unsupervised feature learning, my approach is able to learn a representation useful for the task at hand, thanks purely to the fact that the data used is composed of half graspable and half non-graspable cases.

From Table 2.1, I see that the recognition performance is significantly improved with deep learning methods, improving 9% over the features from [61] and 4.1% over those features combined with FPFH features. Both L_1 and group regularization performed similarly for recognition, but training separate first layer features decreased performance slightly. This shows that learned features, in addition to avoiding hand-design, are able to improve performance significantly over the state of the art. It demonstrates that a deep network is able to learn the concept of "graspability" in a way that generalizes to new objects it hasn't seen before.

Table 2.2 shows that even using any one of the three input modalities (RGB, depth, or surface normals), my algorithm is able to learn features which outperform hand-engineered ones for recognition. Depth gives the highest performance of any single-mode network. Combining depth and normal information improves results over either alone, indicating that they give non-redundant information.

Table 2.3: **Detection results for point and rectangle metrics**, for various learning algorithms.

Algorithm	Image-wise split		Object-wise split	
	Point	Rect	Point	Rect
Chance	35.9	6.7	35.9	6.7
Jiang et al. [61]	75.3	60.5	74.9	58.3
SAE, no mask-based scaling	62.1	39.9	56.2	35.4
SAE, separate layer-1 feat.	70.3	43.3	70.7	40.0
SAE, L_1 reg.	87.2	72.9	**88.7**	71.4
SAE, struct. reg., 1^{st} pass only	86.4	70.6	85.2	64.9
SAE, struct. reg., 2^{nd} pass only	87.5	73.8	87.6	73.2
SAE, struct. reg. two-stage	**88.4**	**73.9**	88.1	**75.6**

The highest accuracy is still obtained by using all the input modalities. This shows that combining depth and color information leads to a system which is more robust than either modality alone. This is due to the fact that some graspable cases (rims of monochromatic objects, etc.) can only be detected using depth information, while in others, the depth channel may be extremely noisy, requiring the use of color information. From this, I can see that integrating multimodal information, a major focus of this work, is important in recognizing good robotic grasps.

Table 2.3 shows that the performance gains from deep learning for recognition carry over to detection, as well. Once mask-based scaling has been applied, all deep learning approaches except for training separate first-layer features outperform the hand-engineered features from [61] by up to 13% for the point metric and 17% for the rectangle metric, while also avoiding the need to design task-specific features. Without mask-based scaling, the system performs poorly, due to the bias illustrated in Fig. 2.5. Separate first-layer features also give weak detection performance, indicating that the relative scores assigned by this form of network are less robust than those learned using my structured regularization approach.

Using structured multimodal regularization also improves results over standard

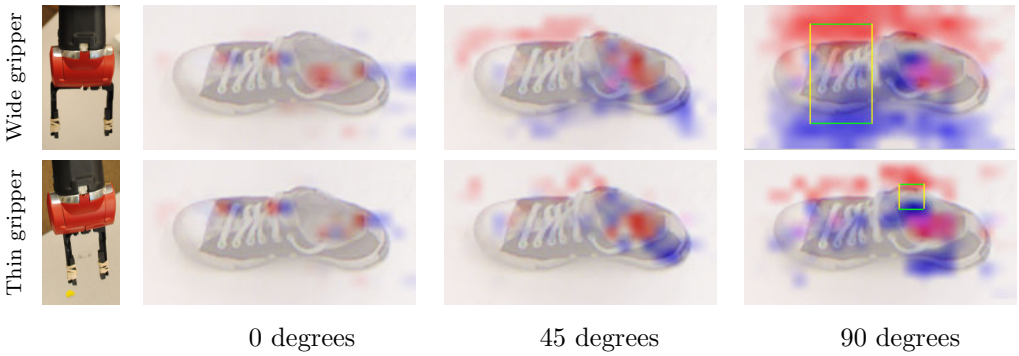


Figure 2.10: **Visualization of grasping scores for different grippers:** Red indicates maximum score for a grasp with left gripper plane centered at each point, blue is similar for the right plate. Best-scoring rectangle shown in green/yellow.

L_1regularization by up to 1.8%, showing that my method also learns more robust features than standard approaches which ignore modality information. Even though using the first-pass network alone underperforms the second-pass network alone by up to 8.3%, integrating both in my two-pass system outperforms the solo second-pass network by up to 2.4%. This shows that the two-pass system improves not only efficiency, but accuracy as well. The performance gains from multimodal regularization and the two-pass system are discussed in detail below.

My system outperforms all baseline approaches by all metrics except for the point metric in the object-wise split case. However, I can see that the chance performance is much higher for the point metric than for the rectangle metric. This shows that the point metric can overstate performance, and the rectangle metric is a better indicator of the accuracy of a grasp detection system.

Adaptability: One important advantage of my detection system is that I can flexibly specify the constraints of the gripper in my detection system. This is particularly important for a robot like Baxter, where different objects might require different gripper settings to grasp. I can constrain the detectors to handle this. Figure 2.10 shows detection scores for systems constrained based on two different

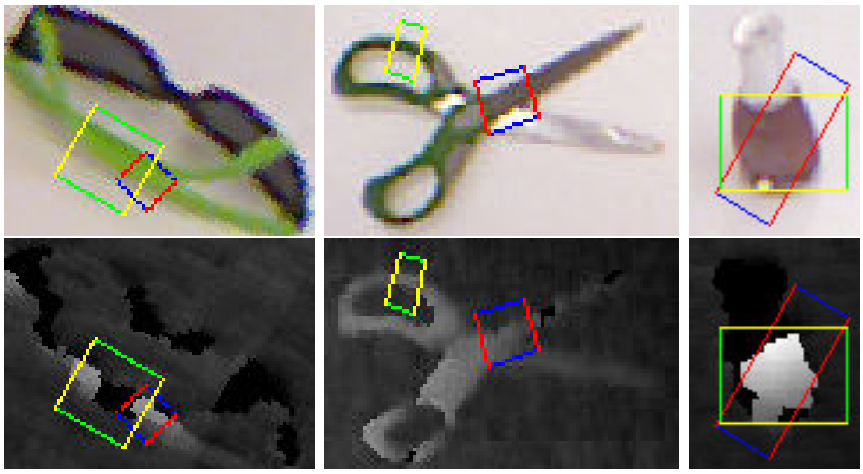


Figure 2.11: **Improvements from group regularization:** Cases where my group regularization approach produces a viable grasp (shown in green and yellow), while a network trained only with simple L_1 regularization does not (shown in blue and red). Top: RGB image, bottom: depth channel. Green and blue edges correspond to gripper.

settings of Baxter's gripper, one wide and one thin. The implications of these results for other types of grippers will be discussed in Section 2.9.

2.7.2 Multimodal Group Regularization

My group regularization term improves detection accuracy over simple L_1 regularization. The improvement is more significant for the object-wise split than for the image-wise split because the group regularization helps the network to avoid overfitting, which will tend to occur more when the learning algorithm is evaluated on unseen objects.

Figure 2.11 shows typical cases where a network trained using my group regularization finds a valid grasp, but a network trained with L_1 regularization does not. In these cases, the grasp chosen by the L_1-regularized network appears valid for some modalities – the depth channel for the sunglasses and nail polish bottle, and the RGB channels for the scissors. However, when all modalities are consid-

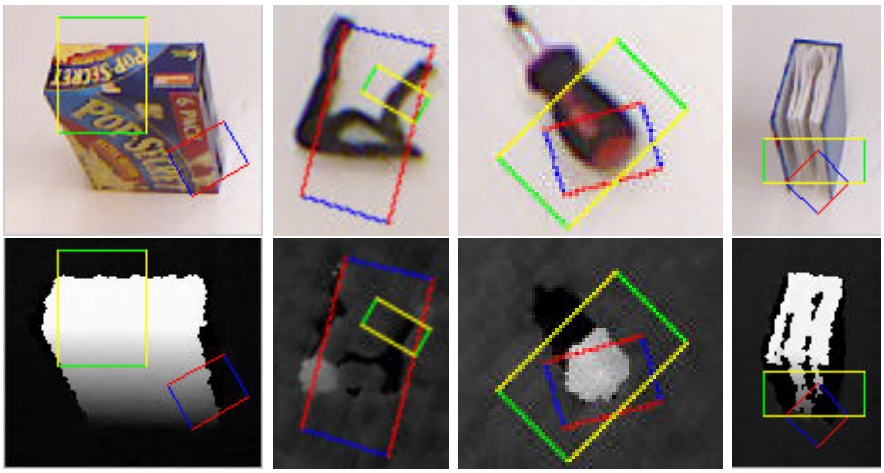


Figure 2.12: **Improvements from two-stage system:** Example cases where the two-stage system produces a viable grasp (shown in green and yellow), while the single-stage system does not (shown in blue and red). Top: RGB image, bottom: depth channel. Green and blue edges correspond to gripper.

ered, the grasp is clearly invalid. The group-regularized network does a better job of combining information from all modalities and is more robust to noise and missing data in the depth channel, as seen in these cases.

2.7.3 Two-stage Detection System

Using my two-pass system enhanced both computational performance and accuracy. The number of rectangles the full-size network needed to evaluate was reduced by roughly a factor of 1000. Meanwhile, detection performance increased by up to 2.4% as compared to a single pass with the large-size network, even though using the small network alone significantly underperforms the larger network. In most cases, the top 100 rectangles from the first pass contained the top-ranked rectangle from an exhaustive search using the second-stage network, and thus results were unaffected.

Figure 2.12 shows some cases where the first-stage network pruned away rectan-

Figure 2.13: **Robotic experiment objects:** Several of the objects used in experiments, including challenging cases such as an oddly-shaped RC car controller, a cloth towel, plush cat, and white ice cube tray.

gles corresponding to weak grasps which might otherwise be chosen by the second-stage network. In these cases, the grasp chosen by the single-stage system might be feasible for a robotic gripper, but the rectangle chosen by the two-stage system represents a grasp which would clearly be successful.

The two-stage system also significantly increases the computational efficiency of my detection system. Average inference time for a MATLAB implementation of the deep network was reduced from 24.6s/image for an exhaustive search using the larger network to 13.5s/image using the two-stage system.

2.8 Robotic Experiments

In order to evaluate the performance of my algorithms in the real world, I ran an extensive series of robotic experiments. To explore the generalizability and

Figure 2.14: **Robots executing grasps:** My robots grasping several objects from the experimental dataset. Top row: Baxter grasping a quad-rotor casing, coffee mug, ice cube tray, knife, and electric shaver. Middle row: Baxter grasping a desk lamp, cheese grater, umbrella, cloth towel, and hot glue gun. Bottom row: PR2 grasping a plush cat, RC car controller, cereal box, toy elephant, and glove.

Table 2.4: **Results for robotic experiments for Baxter,** sorted by object category, for a total of 100 trials. Tr. indicates number of trials, Acc. indicates accuracy (in terms of success percentage.)

Kitchen tools			**Lab tools**			**Containers**			**Toys**			**Others**		
Object	Tr.	Acc.	Object	Tr.	Acc.	Object	Tr.	Acc.	Object	Tr.	Acc.	Object	Tr.	Acc.
Can opener	3	100	Kinect	5	100	Colored cereal box	3	100	Plastic whale	4	75	Electric shaver	3	100
Knife	3	100	Wire bundle	3	100	White cereal box	4	50	Plastic elephant	4	100	Umbrella	4	75
Brush	3	100	Mouse	3	100	Cap-shaped bowl	3	100	Plush cat	4	75	Desk lamp	3	100
Tongs	3	100	Hot glue gun	3	67	Coffee mug	3	100	RC controller	3	67	Remote control	5	100
Towel	3	100	Quad-rotor	4	75	Ice cube tray	3	100	XBox controller	4	50	Metal bookend	3	33
Grater	3	100	Duct tape roll	4	100	Martini glass	3	0	Plastic frog	3	67	Glove	3	100
Average		100	**Average**		90	**Average**		75	**Average**		72	**Average**		85
Overall		84												

Table 2.5: **Results for robotic experiments for PR2,** sorted by object category, for a total of 100 trials. Tr. indicates number of trials, Acc. indicates accuracy (in terms of success percentage.)

Kitchen tools			**Lab tools**			**Containers**			**Toys**			**Others**		
Object	Tr.	Acc.	Object	Tr.	Acc.	Object	Tr.	Acc.	Object	Tr.	Acc.	Object	Tr.	Acc.
Can opener	3	100	Kinect	5	100	Colored cereal box	3	100	Plastic whale	4	75	Electric shaver	3	100
Knife	3	100	Wire bundle	3	100	White cereal box	4	100	Plastic elephant	4	100	Umbrella	4	100
Brush	3	100	Mouse	3	100	Cap-shaped bowl	3	100	Plush cat	4	100	Desk lamp	3	100
Tongs	3	100	Hot glue gun	3	67	Coffee mug	3	100	RC controller	3	67	Remote control	5	100
Towel	3	100	Quad-rotor	4	100	Ice cube tray	3	100	XBox controller	4	25	Metal bookend	3	67
Grater	3	100	Duct tape roll	4	100	Martini glass	3	0	Plastic frog	3	67	Glove	3	100
Average		100	**Average**		95	**Average**		83	**Average**		72	**Average**		95
Overall		89												

effect of the robot on the success rate of my algorithms, I performed experiments on two different robotic platforms, a Baxter Research Robot ("Yogi") and a PR2 ("Kodiak").

Baxter: The first platform used is a Baxter Research Robot, which I call "Yogi." Baxter has two arms with seven degrees of freedom each and a maximum reach of 104 cm, although I used only the left arm for these experiments. The end-effector for this arm is a two-finger parallel gripper. I augmented the gripper tips using rubber bands for additional friction. Baxter's grippers are interchangable, and I used two settings for these experiments - a "wide" setting with an open width of 8 cm and closed width of 4 cm, and a "thin" setting with an open width of 4 cm and a closed width of 0 cm (completely closed, gripper tips touching).

To detect grasps, I mounted a Kinect sensor to Yogi's head, approximately 1.75 m above the ground. angled downwards at roughly a 75^o angle towards a table in front of it. The Kinect gives RGB-D images at a resolution of 640x480 pixels. I calibrated the transformation between the Kinect's and Yogi's coordinate frames by marking four points corresponding to a set of 3D axes, and obtaining the coordinates of these points in both Kinect's and Yogi's frames.

All control for Baxter was done by specifying an end-effector position and orientation, and using the inverse kinematics provided with Baxter to determine a set of joint angles for this pose. Baxter's built-in control systems were used to drive the arm to these new joint angles.

PR2: My second platform was a PR2 robot, "Kodiak." Similar to Baxter, PR2 has two 7-DoF arms with approximately 1 m reach, and I used only the left for these experiments. PR2's grippers open to a width of 8 cm, and are capable of

closing completely from that span, so I did not need to use two settings as with Baxter. I augmented PR2's gripper friction with gaffer tape on the fingertips.

For the experiments on PR2, I used the Kinect already mounted to Kodiak's head, and used ROS's built-in functionality to obtain 3D locations from that Kinect and transform these to Kodiak's body frame for manipulation. Control was performed using the ee_cart stiffness controller [16] with trajectories provided by my own custom MATLAB code.

Experimental Setup: For each experiment, I placed a single object within a 25 cm x 25 cm square on the table, approximately 1.2 m below the mounting point of the Kinect. This square was chosen to be well-contained within each robot's workspace, allowing objects to be reached from most approach vectors. Object positions and orientations were varied between trials, although objects were always placed in configurations in which at least one viable grasp was visible and accessible to the robot.

When using Baxter, due to the limited stroke (span from open to closed) of its gripper, I pre-selected one of the two gripper settings discussed above for each object. I constrained the search space as illustrated in Fig. 2.10 to find grasps for that particular setting.

To detect grasps, I first took an RGB-D image from the Kinect with no objects in the scene as a background image. The depth channel of this image was used to segment objects from the scene, and to correct for the slant of the Kinect. Once an object was segmented, I used my algorithm, as described above, to obtain a single best-ranked grasping rectangle.

The search space for the first-pass network progressed in 15-degree increments

from 15 to 180 degrees (angles larger than 180 being mirror-images of grasps already tested), searching over 10-pixel increments across the image for the X and Y coordinates of the upper-left corner of the rectangle. For the thin gripper setting, rectangle widths and heights from 10 to 40 pixels in 10-pixel increments were searched, while for the thick setting these ranged from 40 pixels to 100 pixels in 20-pixel increments. In both cases, rectangles taller than they were wide were ignored. Once a single best-scoring grasp was detected, I translated it to a robotic grasp consisting of a grasping point and an approach vector using the rectangle's parameters and the surface normal at the rectangle's center as described above.

To execute the grasp, I first positioned the gripper at a location 10 cm back from the grasping point along the approach vector. The gripper was oriented to the approach vector, and rotated around it based on the orientation of the detected grasping rectangle.

Since Baxter's arms are highly compliant, slight imprecisions in end-effector positioning are to be expected – I found that errors of up to 2 cm were typical. Thus, I implemented a visual servoing system using its hand camera, which provides RGB images at a resolution of 320x200 pixels. I used color segmentation to separate the object from the background, and used its lateral position in image space to drive Yogi's end-effector to center the object. I did not implement visual servoing for PR2 because its gripper positioning was found to be precise to within 0.5 cm.

After visual servoing was completed, I drove the gripper 14 cm forwards from its current position along the approach vector, so that the grasping point was well-contained within it. I then closed the gripper, grasping the object, and moved it 30 cm upwards. A grasp was determined to be successful if it was sufficient to lift

the object and hold it for one second.

Objects to be Grasped: For my robotic experiments, I collected a diverse set of 35 objects within a size of .3 m x .3 m x .3 m and weighing at most 2.5 kg (although most were less than 1 kg) from my offices, homes, and lab. Many of them are shown in Fig. 2.13. Most of these objects were not present in the training dataset, and thus were completely new to the grasp detection algorithm.

Due to the physical limitations of the robots' grippers, I found that five of these objects were not graspable even when given a hand-chosen grasp. The small pair of pliers was too low to the table to grip properly. The spray paint can was too smooth for the gripper to get enough friction to lift it. The weight of the hammer was too imbalanced, causing the hammer to rotate and slip out of the gripper when grasped. Similar problems were encountered with the bicycle U-lock. The bevel spatula's handle was too close to the thin-set size of Baxter's gripper, so that I could not position it precisely enough to grasp it reliably. I did not consider these objects for purposes of my experimental results, since my focus was on evaluating the performance of my grasp detection algorithm.

Results: Table 2.4 shows the results of my robotic experiments on Baxter for the remaining 30 objects, a total of 100 trials. Using my algorithm, Yogi was able to successfully execute a grasp in 84% of the trials. Figure 2.14 shows Yogi executing several of these grasps. In 8% of the trials, my algorithm detected a valid grasp which was not executed correctly by Yogi. Thus, I were able to successfully detect a good grasp in 92% of the trials. Video of some of these trials is available at `http://pr.cs.cornell.edu/deepgrasping`.

PR2 yielded a higher success rate as seen in Table 2.5, succeeding in 89% of

trials. This is largely due to the much wider span of PR2's gripper from open to closed and its ability to fully close from its widest position, as well as PR2's ability to apply a larger gripping force. Some specific instances where PR2 and Baxter's performance differed are discussed below.

For comparison purposes, I ran a small set of control experiments for 16 of the objects in the dataset. The control algorithm simply returned a fixed-size rectangle centered at the object's center of mass, as determined by depth segmentation from the background. The rectangle was aligned so that the gripper plates ran parallel to the object's principal axis. This algorithm was only successful in 31% of cases, significantly underperforming my system.

On Baxter, my algorithm sometimes detected a grasp which was not realizable by the current setting of its gripper, but might be executable by others. For example, my algorithm detected grasps across the leg of the plush cat, and the region between the handle and body of the umbrella, both too thin for the wide setting of Baxter's gripper to grasp since it has a minimum span of 4 cm. Since PR2's gripper can close completely from any position, it did not encounter these issues and thus achieved a 100% success rate for both these objects.

The XBox controller proved to be a very difficult object for either robot to grasp. From a top-down angle, there is only a small space of viable grasps with a span of less than 8 cm, but many which have either a slightly larger span (making them non-realizable by either gripper), or are subtly non-viable (e.g. grasps across the two "handles," which tend to slip off.) All viable grasps are very near to the 8 cm span of both grippers, meaning that even slight imprecision in positioning can lead to failure. Due to this, Baxter achieved a higher success rate for the XBox controller thanks to visual servoing, succeeding in 50% of cases as compared to the

25% success rate for PR2.

My algorithm was able to consistently detect and execute valid grasps for a red cereal box, but had some failures on a white and yellow one. This is because the background for all objects in the dataset is white, leading the algorithm to learn features relating white areas at the edges of the gripper region to graspable cases. However, it was able to detect and execute correct grasps for an all-white ice cube tray, and so does not fail for all white objects. This could be remedied by extending the dataset to include cases with different background colors. Interestingly, even though the parameters of grasps detected for the white box were similar for PR2 and Baxter, PR2 was able to succeed in every case while Baxter succeeded only half the time. This is because PR2's increased gripper strength allowed it to execute grasps across corners of the box, crushing it slightly in the process.

Other failures were due to the limitations of the Kinect sensor. I were never able to properly grasp the martini glass because its glossy finish prevented Kinect from returning any depth estimates for it. Even if a valid grasp were detected using color information only, there was no way to infer a proper grasping position without depth information. Grasps for the metal bookend failed for similar reasons, but it was not as glossy as the martini glass, and gave enough returns for some to succeed.

However, my algorithm also had many noteworthy successes. It was able to consistently detect and execute grasps for a crumpled cloth towel, a complex and irregular case which bore little resemblance to any object in the dataset. It was also able to find and grasp the rims of objects such as the plastic baseball cap and coffee mug, cases where there is little visual distinction between the rim and body of the object. These objects underscore the importance of the depth channel

for robotic grasping, as none of these grasps would be detectable without depth information.

My algorithm was also able to successfully detect and execute many grasps for which the approach vector was non-vertical. The grasps shown for the coffee mug, desk lamp, cereal box, RC car controller, and toy elephant shown in Fig. 2.14 were all executed by aligning the gripper to such an approach vector. Indeed, many of these grasps may have failed had the gripper been aligned vertically. This shows that my algorithm is not restricted to detecting top-down grasps, but rather encodes a more general notion of graspability which can be applied to grasps from many angles, albeit within the constraints of visibility from a single-view perspective.

While a few failures occurred, my algorithm still achieved a high rate of accuracy for other oddly-shaped objects such as the quad-rotor casing, RC car controller, and glue gun. For objects with clearly defined handles, such as the cheese grater, kitchen tongs, can opener, and knife, my algorithm was able to detect and execute successful grasps in every trial, showing that there is a wide range of objects which it can grasp extremely consistently.

2.9 Discussion and Future Work

My algorithm focuses on the problem of grasp detection for a two-fingered parallel-plate style gripper. It would be directly applicable to other grippers with fixed configurations, simply requiring new training data labeled with grasps for the gripper in question. My system would allow even the basic features used for grasp detection to adapt to the gripper. This might be useful in cases such as jamming

grippers [62], or two-fingered grippers with differently-shaped contact surfaces, which might require different features to determine a graspable area.

My detection algorithm does not directly address the problem of 3D orientation of the gripper – this orientation is determined only after an optimal rectangle has been detected, orienting the grasp based on the object's surface normals. However, just as my approach here considers aligns a 2D feature window to the gripper, an extension of this work might align a 3D window – using voxels, rather than pixels, as its basic unit of representation for input features to the network. This would allow the system to search across the full 6-DoF 3D pose of the gripper, while still leveraging the power of feature learning.

My system gives only a gripper pose as output, but multi-fingered reconfigurable hands also require a configuration of the fingers in order to grasp an object. In this case, my algorithm could be used as a heuristic to find one or more locations likely to be graspable (similar to the first pass in my two-pass system), greatly reducing the search space needed to find an optimal gripper configuration.

My algorithm also depends only on local features to determine grasping locations. However, many household objects may have some areas which are strongly preferable to grasp over others - for example, a knife might be graspable by the blade, or a hot glue gun by the barrel, but both should actually be grasped by their respective handles. Since these regions are more likely to be labeled as graspable in the data, my system already weakly encodes this, but some may not be readily distinguishable using only local information. Adding a term modeling the probability of each region of the image being a semantically-appropriate area to grasp the object would allow me to incorporate this information. This term could be computed once for the entire image, then added to each local detection score,

keeping detection efficient.

In this work, my visual-servoing algorithm was purely heuristic, simply attempting to center the segmented object underneath the hand camera. However, in future work, a similar feature-learning approach might be applied to hand camera images of graspable and non-graspable regions, improving the visual servoing system's ability to fine-tune gripper position to ensure a good grasp.

Many robotics problems require the use of perceptual information, but can be difficult and time-consuming to engineer good features for, particularly when using RGB-D data. In future work, my approach could be extended to a wide range of such problems. My system could easily be applied to other detection problems such as object detection or obstacle detection. However, it could also be adapted to other similar problems, such as object tracking and visual servoing.

Multimodal data has become extremely important for robotics, due both to the advent of new sensors such as the Kinect and the application of robots to more challenging tasks which require multiple modalities of information to perform well. However, it can be very difficult to design features which do a good job of integrating many modalities. While my work focuses on color, depth, and surface normals as input modes, my structured multimodal regularization algorithm might also be applied to others. This approach could improve performance while allowing roboticists to focus on other engineering challenges.

2.10 Conclusions

I presented a system for detecting robotic grasps from RGB-D data using a deep learning approach. My method has several advantages over current state-of-the-art methods. First, using deep learning allows me to avoid hand-engineering features, learning them instead. Second, my results show that deep learning methods significantly outperform even well-designed hand-engineered features from previous work.

I also presented a novel feature learning algorithm for multimodal data based on group regularization. In extensive experiments, I demonstrated that this algorithm produces better features for robotic grasp detection than existing deep learning approaches to multimodal data. My experiments and results, both offline and on real robotic platforms, show that my two-stage deep learning system with group regularization is capable of robustly detecting grasps for a wide range of objects, even those previously unseen by the system.

CHAPTER 3

DEEPMPC: LEARNING DEEP LATENT FEATURES FOR MODEL PREDICTIVE CONTROL

3.1 Introduction

As robots perform tasks in the real world, they must be able to handle a large variety of environments, objects, materials, and more. Traditional robotics approaches which hand-code controllers and models are ill-equipped to deal with this, both because it is time-consuming to create controllers for all such cases, and because a human programmer cannot possibly anticipate the large variety of situations that may be encountered.

Most real-world tasks involve interactions with complex, non-linear dynamics. Although practiced humans are able to control these interactions intuitively, developing robotic controllers for them is very difficult. Several common household activities fall into this category, including scrubbing surfaces, folding clothes, interacting with appliances, and cutting food. Other applications include surgery, assembly, and locomotion. These interactions are characterized by hard-to-model effects, involving friction, deformation, and hysteresis. The compound interaction of materials, tools, environments, and manipulators further alters these effects. Consequently, the design of controllers for such tasks is highly challenging.

In recent years, "feed-forward" model-predictive control (MPC) has proven effective for many complex tasks, including quad-rotor control [132], mobile robot

This work originally presented as a conference paper at Robotics: Science and Systems (RSS) 2015, and is under submission to the International Journal of Robotics Research (IJRR). This was joint work with Ross Knepper and Ashutosh Saxena.

maneuvering [51], full-body control of humanoid robots [38], and many others [88, 49, 33]. The key insight of MPC is that an accurate predictive model allows me to optimize control inputs for some cost over both inputs and predicted *future* outputs. Such a cost function is often easier and more intuitive to design than completely hand-designing a controller. The chief difficulty in MPC lies instead in designing an accurate dynamics model.

Let us consider the dynamics involved in cutting food items, as shown in Fig. 3.1, for the wide range of materials shown in Fig. 3.2. An effective cutting strategy depends heavily on properties of the food, including its coefficient of friction with the knife, elastic modulus, fracture effects, and hysteretic effects such as plastic deformation [96]. These variations lead humans to such diverse cutting strategies as slicing, sawing, and chopping. In fact, properties can even vary within a single material – compare cutting through the skin of a lemon to cutting its flesh. Thus, a major challenge of this work is to design a model which can estimate and make use of global environmental properties such as the material and tool in question and temporally-changing properties such as the current rate of motion, depth of cutting, enclosure of the knife by the material, and layer of the material the knife is in contact with. While some works [42] attempt to define parameters modeling these properties, it is very difficult to design a set that truly captures all these complex inter- and intra-material variations.

Developing a model for such a complex task is thus extremely challenging. The model must be able to handle a wide range of non-linear dynamics. It must also be able to model the effects of a huge range of properties and variations on these effects, and infer these properties online while acting. They must be able to model the entire range of variations in dynamics the model might see in the

Figure 3.1: **Cutting food:** My PR2 robot uses my algorithms to perform complex, precise food-cutting operations. Given the large variety of material properties, it is challenging to design appropriate controllers.

real world, extremely challenging to do with hand-defined properties. In order to be useful for MPC, the model's outputs must be differentiable with respect to its inputs, and both forward prediction and backwards gradient computation must be time-efficient in order to allow real-time optimization of the control inputs.

For these reasons, I take a deep learning approach. In the recent past, such methods have proven effective for learning latent task-specific features across many domains [7, 50, 79, 27, 95, 58, 140]. In this chapter, I give a novel deep architecture for physical prediction for complex tasks such as food cutting. When this model is used for predictive control, it yields a DeepMPC controller which is able to learn task-specific controls.

Since deep networks can act as universal function approximators [7], they can model even complex, non-linear dynamics. I make use of conditional multiplicative structures [91] to model the dependence of these dynamics on different properties. I treat these properties as latent, allowing the model itself to learn features which are useful for physical prediction. I apply temporal recurrence [119] to allow the model to reuse previous information both to refine estimates of these properties and to model temporal variation in properties. Deep networks are an excellent choice as a model for real-time MPC because they are easily and efficiently differentiable with respect to their inputs using the same back-propagation algorithms used in learning, and because network sizes can simply be adjusted to trade off between prediction accuracy and computational speed.

My model, optimized for receding-horizon prediction, learns latent material properties directly from data. My architecture uses multiplicative conditional interactions and temporal recurrence to model both inter-material and time-dependent intra-material variations. I also present a novel learning algorithm for this recurrent network which avoids overfitting and the "exploding gradient" problem commonly seen when training recurrent networks [8]. Once learned, inference for my model is extremely fast - when predicting to a time-horizon of 1s (100 samples) in the future, my model and its gradients can be evaluated at a rate of 1.2kHz.

In extensive experiments on my large-scale dataset comprising 1488 examples of robotic cutting across 20 different material types, I demonstrate that my feature-learning approach outperforms other state-of-the-art methods for physical prediction. I also implement an online real-time model-predictive controller using my model. In a series of over 450 real-world robotic trials, I show that my controller

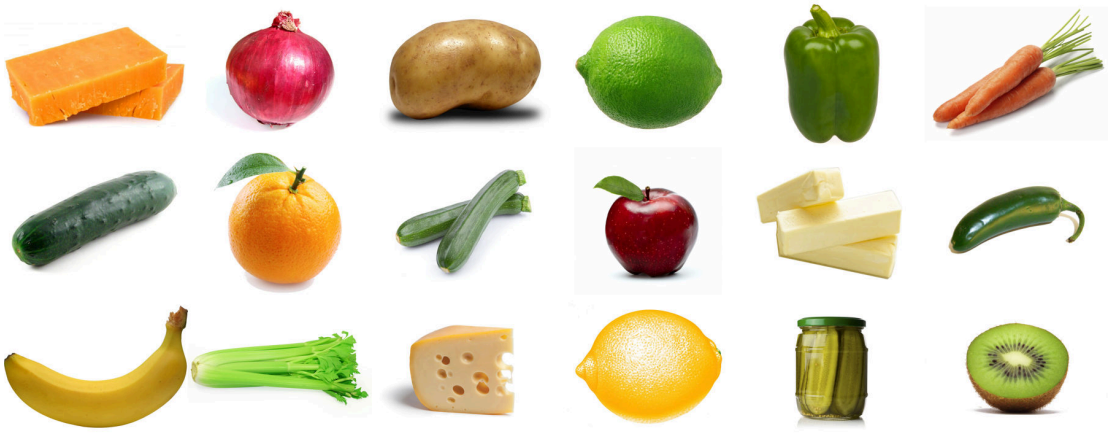


Figure 3.2: **Food materials:** Some of the 20 diverse food materials which make up my material interaction dataset. These include tough vegetables like carrots and potatoes, thick-skinned fruits like lemons and limes, and soft items like butter and bananas, all of which require different techniques to cut properly.

gives extremely strong performance for robotic food-cutting, even compared to methods tuned for specific material classes.

In summary, the contributions of this chapter are:

- I combine deep learning and model-predictive control in a DeepMPC controller that uses learned task dynamics.
- I propose a novel deep architecture which is able to model dynamics conditioned on learned latent properties and a multi-stage pre-training algorithm that avoids common problems in training recurrent neural networks.
- I implement a real-time model predictive control system using my learned dynamics model on a PR2 robot.
- I demonstrate that my model and controller give strong performance for the difficult task of robotic food-cutting.

The remainder of this chapter is organized as follows: I present related work, including an overview of model learning for control and related methods in deep

learning in Section 3.2. I then introduce and define the food-cutting problem and model predictive control in Section 3.3. I motivate and present my deep architecture for modeling complex, varying dynamics in Section 3.4, then present my learning and inference algorithms for it in Section 3.5. I give other system details in Section 3.6, then present my real-time MPC implementation on a PR2 robot in Section 3.7. I present experiments and results for my approach for modeling the complex dynamics involved in food-cutting in Section 3.8, and for real-world robotic control on a PR2 robot in Section 3.9. Finally, I conclude and present directions for future work in Section 3.10.

3.2 Related Work

3.2.1 Robotic Control

Reactive feedback controllers, where a control signal is generated based on error from current state to some set-point, have been applied since the 19^{th} century [9]. Stiffness control, where error in robot end-effector pose is used to determine end-effector forces, remains a common technique for compliant, force-based activities [16, 6, 42]. Such approaches are limited because they require a trajectory to be given beforehand, making it difficult to adapt to different conditions.

Markov Decision Processes (MDPs) [114] are another popular approach to robotic control. While these methods give a tractable, general approach to solving many robotic problems such as autonomous helicopter flight [1], robotic soccer [116] and many others, they are limited to problems with discrete, fully-observable states. In my food-cutting problem and many other robotic problems, I are deal-

ing with continuous states (physical positions), and some environmental properties (e.g. material types) may not be directly observable. Partially-Observable MDPs (POMDPs), which have been applied to such diverse problems as action anticipation for table tennis [147], robotic grasping [53], navigation [40], avoid these assumptions. However, they typically still make others, such as locally linear dynamics [18] or discrete action spaces [42]. In this work, both my states and actions will be continous-valued, and I will directly model the fact that task dynamics depend on some unobserved properties.

Feed-forward model-predictive control allows controls to adapt online by optimizing some cost function over predicted future states. These approaches have gained increased attention as modern computing power makes it feasible to perform optimization in real time. Shim et al. [132] used MPC to control multiple quad-rotors in simulation, while Howard et al. [51] performed intricate maneuvers with real-world mobile robots. Erez et al. [38] used MPC for full-body control of a humanoid robot. These approaches have been extended to many other tasks, including underwater vehicle control [88], visual servoing [49], and even heart surgery [33]. However, all these works assume the task dynamics model is fully specified.

3.2.2 Model Learning for Control

Model learning for robot control has also been a very active area, and I refer the reader to a review of work in the area by Nguyen-Tuong and Peters [106]. While early works in model learning [2, 103] fit parameters of some hand-designed task-specific model to data, such models can be difficult to design and may not generalize well to new tasks. Thus, several recent works attempt to learn more general dynamics models such as Gaussian mixture models [21, 66] and Gaussian

processes [68]. Neural networks [22, 19] are another common choice for learning general non-linear dynamics models. The highly parameterized nature of these models allows them to fit a wide variety of data well, but also makes them very susceptible to overfitting.

3.2.3 Policy Learning

Several recent approaches to control learning first learn a dynamics model, then use this model to learn a policy which maps from system state to control inputs. These works often iteratively use this policy to collect more data and re-learn a new policy in an online learning approach. Levine et al. [82] use a Gaussian mixture model (GMM) where linear models are fit to each cluster, while Deisenroth and Rasmussen [30] use a Gaussian process (GP.) Experimentally, both these models gave less accurate predictions than mine for robotic food-cutting. The GP also had very long inference times (roughly 10^6 times longer than mine) due to the large amount of training data needed. For details, see Section 3.8. This weak performance is because they use only temporally-local information, while my model uses learned recurrent features to integrate long-term information and model unobserved system properties such as materials.

These works focus on online policy search, while here I focus on modeling and application to real-time MPC. My model could be used along with them in a policy-learning approach, allowing them to model dynamics with environmental and temporal variations. However, my model is efficient enough to optimize for predictive control at run-time. This avoids the possibility of learned policies overfitting the training data and allows the cost function and its parameters to be changed online. It also allows my model to be used with other algorithms which

use its predictions directly.

3.2.4 Robotic Manipulation

Luo and Hauser [84] developed a system which adapts manipulation to unknown system parameters, but requires a parameterized dynamics model. Koval et al. [70] developed a new algorithm for planar contact manipulation which decomposes pre- and post-contact policies. Maitin-Shepard et al. [86] developed a system for robotic towel-folding. This system focused on the perception aspects of the problem, and assumes uniformity and compliance in the material being manipulated.

Several recent works have applied robotic manipulation to kitchen operations. Bollini et al. [16] developed a vision-based robotic system for preparing and baking cookies, while Beetz et al. [6] developed a system for preparing pancakes. Gemici and Saxena [42] presented a learning system for manipulating deformable objects which infers a set of material properties, then uses these properties to map objects to a latent set of haptic categories which are used to determine how to manipulate the object. However, their approach requires a predefined set of properties (*plasticity*, *brittleness*, etc.), and chooses between a small set of discrete actions. By contrast, my approach performs continuous-space real-time control, and uses *learned* latent features to model material properties and other variations, avoiding the need for hand-design. All three of these works also apply non-reactive stiffness controllers.

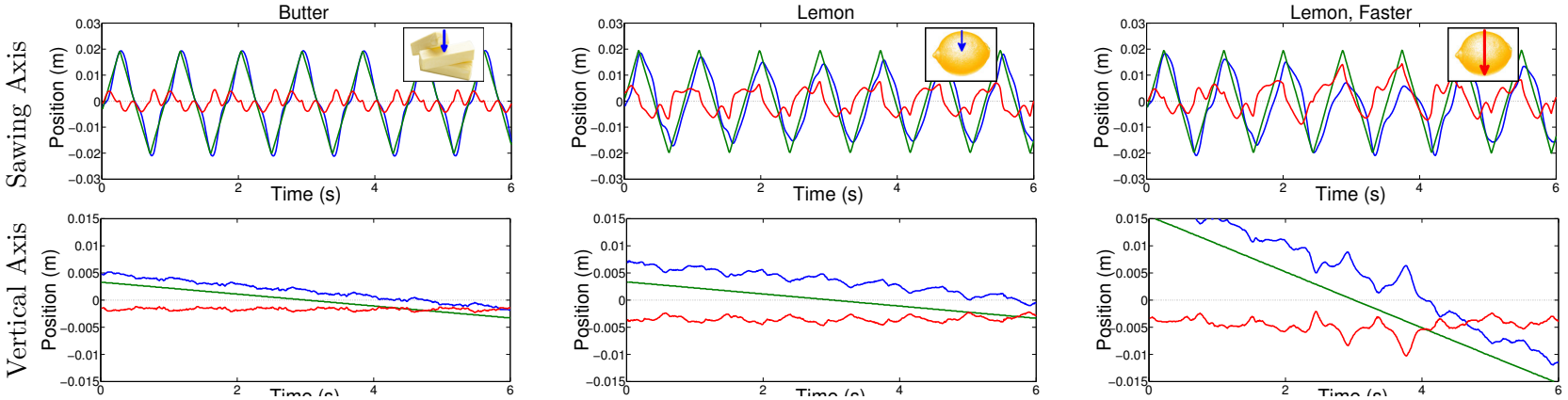


Figure 3.3: **Variation in cutting dynamics:** plots showing desired (green) and actual (blue) trajectories, along with error (red) obtained using a stiffness controller while cutting butter (left) and a lemon at low (middle) and high (right) rates of vertical motion. Butter resists the knife significantly less than the lemon. Even though only the vertical cutting rate is the only change between the middle and right-hand plots, dynamics along the sawing axis also change significantly. Dynamics also vary with time for the lemon as the knife cuts through the skin and into the flesh.

3.3 Problem Definition and System

In this work, I focus on the task of cutting a wide range of food items. This problem is a good testbed for my algorithms because of the variety of dynamics involved in cutting different materials. Designing individual controllers for each material would be very time-consuming, and hand-designing accurate dynamics models for each would be nearly infeasible.

For the task of cutting, I define gripper axes as shown in Fig. 3.4, such that the X axis points out of the point of the knife, Y axis normal to the blade, and Z axis vertically. Here, I consider linear cutting, where the goal is to make a cut of some given length along the Z axis. The control inputs to the system are denoted as $u^{(t)} = (F_x^{(t)}, F_y^{(t)}, F_z^{(t)})$, where $F_x^{(t)}$ represents the force, in Newtons, applied along the end-effector X axis at time t. The physical state of the system is $x^{(t)} = (P_x^{(t)}, P_y^{(t)}, P_z^{(t)})$ where $P_x^{(t)}$ is the X coordinate of the end-effector's position at time t.

A simple approach to control for this problem might use a fixed-trajectory stiffness controller, where control inputs are proportional to the difference between the current state $x^{(t)}$ and some desired state $x^{*(t)}$ taken from a given trajectory.

Fig. 3.3 shows some examples which demonstrate the difficulties inherent in this approach. While some materials, such as the butter shown on the left, offer very little resistance, allowing a stiffness controller to accurately follow a given trajectory, others, such as the lemon shown in the remaining two plots, offer more resistance, causing significant deviation from the desired trajectory. When cutting a lemon, I can also see that the dynamics change with time, resisting the knife more as it cuts through the skin, then less once it enters the flesh of the lemon.

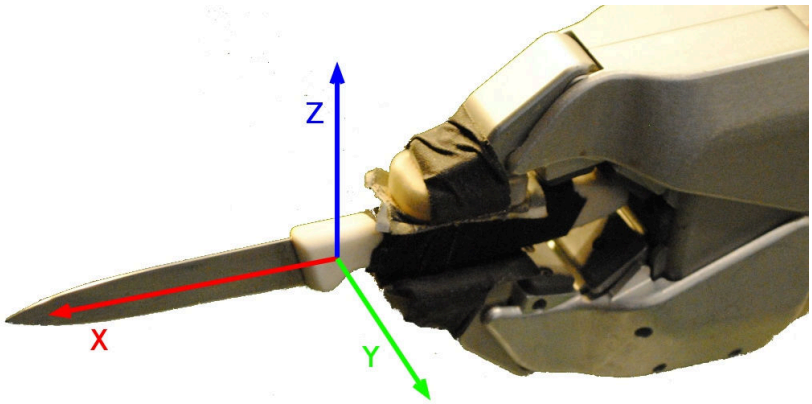


Figure 3.4: **Gripper axes:** PR2's gripper with knife grasped, showing the axes used in this chapter. The X ("sawing") axis points along the blade of the knife, Y points normal to the blade, and Z points vertically.

The dynamics of the sawing and vertical axes are also coupled - increasing the rate of vertical motion increases error along the sawing axis, even though the same controls are used for that axis. This coupled behavior presents additional challenges for modeling and control, as these axes must be considered together.

In my approach, I fix the orientation of the end-effector, as well as the position of the knife along its Y axis, using stiffness control to stabilize these. However, even though my primary goal is to move the knife along its Z axis, as shown in Fig. 3.3, the X and Z axes are strongly coupled for this problem. Thus, my algorithm performs control along both the X and Z axes. This allows "sawing" and "slicing" motions in which movement along the X axis is used to break static friction along the Z axis and enable forward progress. A nonlinear function f predicts future states:

$$\hat{x}^{(t+1)} = f(x^{(t)}, u^{(t+1)}) \tag{3.1}$$

This formula can then be applied recurrently to predict further into the future, e.g. $\hat{x}^{(t+2)} = f(\hat{x}^{(t+1)}, u^{(t+2)})$. When performing recurrent prediction as such, an accurate dynamics model is extremely important as errors will accumulate over

multiple timesteps.

3.3.1 Model-Predictive Control: Background

In this work, I use a model-predictive controller to control the cutting hand. Such controllers have been shown to work extremely well for a wide variety of tasks for which hand-defined controllers are either difficult to define or simply cannot suffice [51, 38, 88, 33]. Defining $X_{t:k}$ as the system state from time t through time k, and $U_{t:k}$ similarly for system inputs, a model-predictive controller works by finding a set of optimal inputs $U^*_{t+1:t+T}$ which minimize some cost function $C(\hat{X}_{t+1:t+T}, U_{t+1:t+T})$ over predicted state $\hat{X}$ and control inputs U for some finite time horizon T:

$$U^*_{t+1:t+T} = \arg\max_{U_{t+1:t+T}} C(\hat{X}_{t+1:t+T}, U_{t+1:t+T}) \tag{3.2}$$

This approach is powerful, as it allows the robot to leverage its knowledge of task dynamics $f(x, u)$ directly, predicting future interactions and proactively avoiding mistakes rather than simply reacting to past observations. It is also versatile, as I have the freedom to design C to define optimality for some task. The chief difficulty lies in modeling the task dynamics $f(x, u)$ in a way that is both differentiable and quick to evaluate, to allow online optimization.

3.4 Modeling Time-Varying Non-Linear Dynamics with Deep Networks

Hand-designing models for the entire range of potential interactions encountered in complex tasks such as cutting food would be nearly impossible. My main challenge

in this work is then to design a model capable of *learning* non-linear, time-varying dynamics. This model must be able to respond to short-term changes, such as breaking static friction, and must be able to identify and model variations caused by varying materials and other properties. It must be differentiable with respect to system inputs, and the system outputs and gradients must be fast to compute to be useful for real-time control.

I choose to base my model on deep learning algorithms, a strong choice for my problem for several reasons. They have been shown to be general non-linear learners [7], but remain differentiable using efficent back-propagation algorithms. When time is an issue, as in my case, network sizes can be scaled down to trade accuracy for computational performance.

Although deep networks can learn any non-linear function, care must still be taken to design a task-appropriate model. As shown in Fig. 3.7, a simple deep network gives relatively weak performance for this problem. Thus, one major contribution of this work is to design a novel deep architecture for modeling dynamics which might vary both with the environment, material, etc., and with time while acting. In this section, I describe my architecture, shown in Fig. 3.5 and motivate my design decisions in the context of modeling such dynamics.

3.4.1 Deep Learning - Background

[Move this up to a new DL overview section in global intro]

Before describing my new architecture for handling complex, varying dynamics, I will first describe previous work which makes up some components of this architecture and algorithm.

Unsupervised Feature Learning: Unsupervised feature learning is one of the major strengths of modern deep learning approaches. Even for unlabled data, these algorithms are capable of learning useful features which can be used to initialize the network before supervised learning. Since they are generic to the type of data used as input, they can be applied to learned features to learn multiple layers of representation. Here, I will apply a variant of the sparse auto-encoder (SAE) algorithm [46], which learns features which reconstruct the training data well while activating sparsely (e.g. for a given case, only a few features should have high values.) Initializing the network in this way helps to avoid overfitting by giving supervised learning a better, more general starting point.

Back-propagation: During both supervised fine-tuning and some gradient-based unsupervised learning algorithms such as SAE, I use back-propagation to efficiently compute cost function gradients with respect to each parameter of the network. This works by first performing forward inference in the network, then computing the cost function and its gradient with respect to its inputs (from the network.) We can then iteratively propagate this gradient through the network, computing the gradient of the cost function with respect to each hidden unit and weight.

A similar approach can be used to compute cost function gradients with respect to system inputs during online MPC. Here, gradients with respect to network parameters are irrelevant, so I propagate cost function gradients backwards through each layer of hidden features until they reach the system inputs. This lets me quickly and efficiently compute the gradient of some cost function over network outputs with respect to system inputs, allowing me to perform real-time gradient-based optimization for MPC.

Conditional Features: When modeling dynamics which depend on the robot's environment, I want to be able to condition the current dynamic response on some set of environmental properties. Factored conditional nodes [91, 143] are able to model conditional structures such as this by learning weights from each set of inputs to some hidden "factors." Each factor then multiplies all inputs it recieves, effectively scaling the contribution of each set of inputs based on features extracted from the others. When modeling dynamics, this is useful as it allows different dynamics to be activated or deactivated depending on environmental properties. Such features have also been shown to be useful in detecting transformations, such as shifts and rotations in natural images [91], a behavior which I will use when modeling time-varying properties.

Temporal Recurrence: When modeling time-dependent behavior, it is useful to re-use past information and features. Temporal recurrence [119] allows us to do so by forming weights to a set of features from the same features for the previous timestep, and, in turn, feeding these features forwards to the next. This allows us to naturally re-use features and integrate long-term information, while still allowing features to change over time. This is also memory-efficient, as the system need only remember one additional set of features (the previous timestep's) and do not need to record all observed features.

However, recurrent models introduce additional difficulty during training, in particular in cases such as recurrent dynamic prediction where the model outputs are also used recurrently. If the network is not initialized well, inaccurate predictions will be fed forwards, causing increased inaccuracy in future timesteps, leading to the "exploding gradient" problem when already-huge error gradients are further scaled up during back-propagation [8]. Such gradients cause problems for

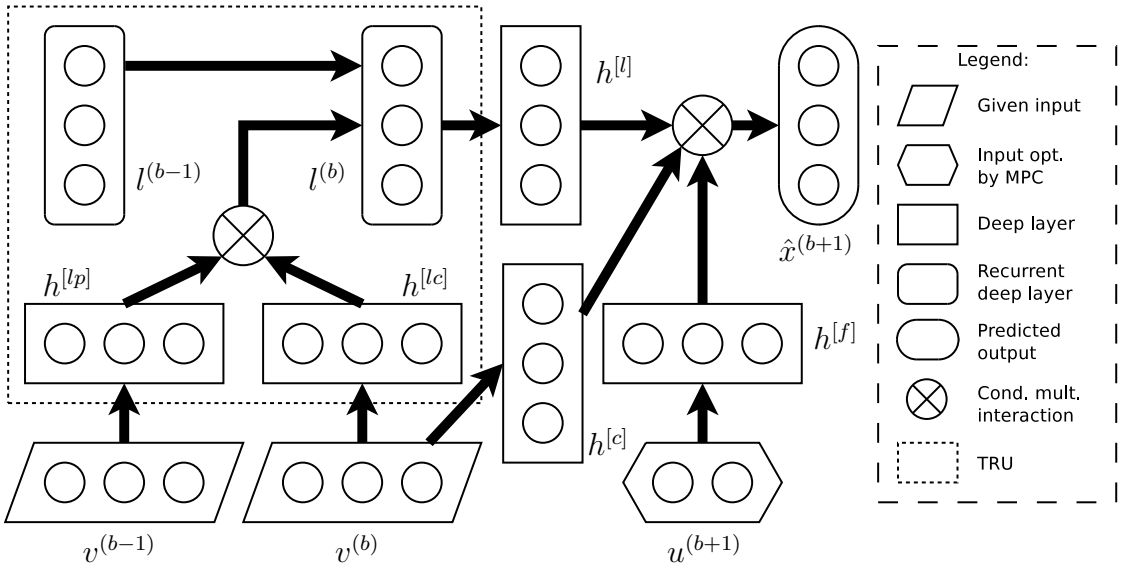


Figure 3.5: **Deep predictive model:** Architecture of my recurrent conditional deep predictive dynamics model. Transforming recurrent units (TRUs) on the left model time-varying latent properties which affect system dynamics. On the right, conditional multiplicative modulation is used again to condition future system responses on past observed dynamics and latent features.

gradient-based learning algorithms and can often lead to overfitting the training data. Thus, in this work, I present a new algorithm for training my recurrent model for physical prediction which iteratively initializes portions of it to avoid these issues.

3.4.2 DeepMPC Architecture

In order to properly model complex dynamics conditioned on environmental properties which might vary with time, I now define a new deep architecture. This architecture retains the strengths of standard deep learning algorithms while giving superior predictive performance for complex tasks such as food cutting, as shown in Section 3.8.

Dynamic Response Features: When modeling physical dynamics, it is important to capture short-term input-output responses. Thus, rather than learning features separately for system inputs u and outputs x, the basic input features used in my model are a concatenation of both. It is also important to capture high-order and delayed-response modes of interaction. Thus, rather than considering only a single timestep, I consider blocks thereof when computing these features, so that for block b, with block size B, I have visible input features $v^{(b)} = (X_{b*B:(b+1)*B-1}, U_{b*B:(b+1)*B-1})$. For known timesteps, I use the observed values of x, while for future timesteps, I use $\hat{x}$ as predicted by my model. For more details on my feature pre-processing, see Section 3.6.

Conditional Dynamic Responses: For tasks such as material cutting, local dynamics might be conditioned on both time-invariant and time-varying properties. Thus, I must design a model which operates conditional on past information. I do so by introducting factored conditional units [91], where features from some number of inputs modulate multiplicatively and are then weighted to form network outputs. Intuitively, this allows me to blend a set of models based on features extracted from past information. Since my model needs to incorporate both short- and long-term information, I allow three sets of features to interact – the current control inputs, the past block's dynamic response, and latent features modeling long-term observations, described below. Although the past block's response is also included when forming latent features, including it directly in this conditional model frees my latent features from having to model such short-term dependencies.

I use c to denote the current timeblock, f to denote the immediate future one, l for latent features, and o for outputs. Take N_v as the number of features v, N_x as the number of states x, and N_u as the number of inputs u in a block, and N_l

as the number of latent features l. With $h^{[c](b)} \in \mathbb{R}^{N_{oh}}$ as the hidden features from the current timestep, formed using weights $W^{[c]} \in \mathbb{R}^{N_v \times N_{oh}}$ (similar for f and l), and $W^{[o]} \in \mathbb{R}^{N_{oh} \times N_x}$ as the output weights, my predictive model is then:

$$h_j^{[c](b)} = \sigma\left(\sum_{i=1}^{N_v} W_{i,j}^{[c]} v_i^{(b)}\right) \tag{3.3}$$

$$h_j^{[f](b)} = \sigma\left(\sum_{i=1}^{N_u} W_{i,j}^{[f]} u_i^{(b+1)}\right) \tag{3.4}$$

$$h_j^{[l](b)} = \sigma\left(\sum_{i=1}^{N_l} W_{i,j}^{[l]} l_i^{(b)}\right) \tag{3.5}$$

$$\hat{x}_j^{(b+1)} = \sum_{i=1}^{N_{oh}} W_{i,j}^{[o]} h_i^{[c](b)} h_i^{[f](b)} h_i^{[l](b)} \tag{3.6}$$

Long-Term Recurrent Latent Features: Another major challenge in modeling time-dependent dynamics is integrating long-term information while still allowing for transitions in dynamics, such as moving from the skin to the flesh of a lemon. To this end, I introduce transforming recurrent units (TRUs). To retain state information from previous observations, my TRUs use temporal recurrence, where each latent unit has weights to the previous timestep's latent features. To allow this state to transition based on locally observed transformations in dynamics, they use the paired-filter behavior of multiplicative interactions to detect transitions in the dynamic response of the system and update the latent state accordingly. In previous work, multiplicative factored conditional units have been shown to work well in modeling transformations in images [91] and physical states [143], making them a good choice here. Each TRU thus takes input from the previous TRU's output and the short-term response features for the current and previous time-blocks. With ll denoting recurrent weights, lc denoting current-step for the latent features, lp previous-step, and lo output, and N_{lh} as the number of

TRU hidden units, my latent feature model is then:

$$h_j^{[lc](b)} = \sigma\left(\sum_{i=1}^{N_v} W_{i,j}^{[c]} v_i^{(b)}\right) \tag{3.7}$$

$$h_j^{[lp](b)} = \sigma\left(\sum_{i=1}^{N_v} W_{i,j}^{[f]} v_i^{(b-1)}\right) \tag{3.8}$$

$$l_j^{(b)} = \sigma\left(\sum_{i=1}^{N_{lh}} W_{i,j}^{[lo]} h_i^{[lc](b)} h_i^{[lp](b)} + \sum_{k=1}^{N_l} W_{k,j}^{[ll]} l_k^{(b-1)}\right) \tag{3.9}$$

Finally, Fig. 3.5 shows the full architecture of my deep predictive model, as described above.

3.5 Learning and Inference

In this section, I define the learning and inference procedures for the model defined above. The online inference approach is a direct result of my model. However, there are many possible approaches to learning its parameters. Neural networks require a huge number of parameters (weights) to be learned, making them particularly susceptible to overfitting, and recurrent networks often suffer from instability in future predictions, causing large gradients which make optimization difficult (the "exploding gradient" problem [8]).

To avoid these issues, I define a new three-stage learning approach which pre-trains the network weights before using them for recurrent modeling. Deep learning methods are non-convex, and converge to only a *local* optimum, making my approach important in ensuring that a good optimum which does not overfit the training data is reached.

Inference: During inference for MPC, we are currently at some time-block

b with latent state $l^{(b)}$, known system state $x^{(b)}$ and control inputs $u^{(b)}$. Future control inputs $U_{t+1:t+T}$ are also given, and the goal is then to predict the future system states $\hat{X}_{t+1:t+T}$ up to time-horizon T, along with the gradients $\partial X/\partial U$ for all pairs of x and u. These gradients will then be used by MPC to iteratively optimize the control inputs u.

I perform this inference by applying my model recurrently to predict future states up to time-horizon T, using predicted states $\hat{x}$ and latent features l as inputs to my predictive model for subsequent timesteps, e.g. when predicting $x^{(b+2)}$, I use the known $x^{(b)}$ along with the predicted $\hat{x}^{(b+1)}$ and $l^{(b+1)}$ as inputs.

My model's outputs ($\hat{x}$) are differentiable with respect to all its inputs, allowing me to take gradients $\partial X/\partial U$ using an approach similar to the backpropagation-through-time algorithm used to optimize model parameters during learning, as shown in Algorithm 1. I can in turn use these gradients with any gradient-based optimization algorithm to optimize $U_{t+1:t+T}$ with respect to some differentiable cost function $C(X, U)$. No online optimization is necessary to perform inference for my model. For details on my online optimization approach, see Section 3.7.

Learning: During learning, my objective is to use my training data to learn a set of model parameters $\Theta = (W^{[f]}, W^{[c]}, W^{[l]}, W^{[o]}, W^{[lp]}, W^{[lc]}, W^{[ll]}, W^{[lo]})$ which minimize prediction error while avoiding overfitting.

A naive approach to learning might randomly initialize Θ, then optimize the entire recurrent model for prediction error. However, random weights would likely cause the model to make inaccurate predictions, which will in turn be fed forwards to future timesteps. This could cause huge errors at time-horizon T, which will in turn cause large gradients to be back-propagated, resulting in instability in the

learning and overfitting to the training data. To remedy this, I propose a multi-stage pre-training approach which first optimizes some subsets of the weights, leading to much more accurate predictions and less instability when optimizing the final recurrent network. I show in Fig. 3.7 that my learning algorithm significantly outperforms random initialization.

Phase 1: Unsupervised Pre-Training: In order to obtain a good initial set of features for l, I apply an unsupervised learning algorithm similar to the sparse auto-encoder algorithm [46] to train the non-recurrent parameters of the TRUs. This algorithm first projects from the TRU inputs up to l, then uses the projected l to reconstruct these inputs. The TRU weights are optimized for a combination of reconstruction error and sparsity in the outputs of l. Taking $v^{m,k}$ as the visible features for the k^{th} time-block of training case m, M as the number of training cases, and T_m as the number of timesteps for case m, and $g(l)$ as some function penalizing latent feature activation to induce sparsity, my unsupervised pre-training phase proceeds as:

$$\Theta^* = \arg\min_{\Theta} \sum_{m=1}^{M} \sum_{b=2}^{T_m/B} ||\hat{v}^{(m,b-1)} - v^{(m,b-1)}||_2^2 + ||\hat{v}^{(m,b)} - v^{(m,b)}||_2^2 + \lambda \sum_{j=1}^{K} g(l_j^{(b)}) \tag{3.10}$$

$$h_j^{[ll](b)} = \sum_{a=1}^{N_l} W_{j,a}^{[ll]} l_a^{(b)} \tag{3.11}$$

$$\hat{v}_i^{(b-1)} = \sum_{j=1}^{N_{lh}} W_{i,j}^{[lp]} h_j^{[ll](b)} h_j^{[lc](b)} \tag{3.12}$$

$$\hat{v}_i^{(t)} = \sum_{j=1}^{N_{lh}} W_{i,j}^{[lp]} h_j^{[ll](b)} h_j^{[lp](b)} \tag{3.13}$$

Phase 2: Short-term Prediction Training: While I could now use these parameters as a starting point to optimize a fully recurrent multi-step prediction system, I found that in practice, this lead to instability in the predicted values, since inaccuracies in initial predictions might "blow up" and cause huge deviations in future timesteps.

Instead, I include a second pre-training phase, where I train the model to predict a single timestep into the future. This allows the model to adjust from the task of reconstruction to that of physical prediction, without risking the aforementioned instability. For this stage, I remove the recurrent weights from the TRUs, effectively setting all $W^{[ll]}$ to zero and ignoring them for this phase of optimization.

Taking $x^{(m,k)}$ as the state for the k^{th} time-block of training case m, M as the number of training cases, and B_m as the number of timeblocks for case m, this stage optimizes:

$$\Theta^* = \arg\min_{\Theta} \sum_{m=1}^{M} \sum_{b=2}^{B_m - 1} ||\hat{x}^{(m,b+1)} - x^{(m,b+1)}||_2^2 \tag{3.14}$$

Phase 3: Warm-Latent Recurrent Training: Once Θ has been pre-trained by these two phases, I use them to initialize a recurrent prediction system which performs inference as described above. I then optimize this system to minimize the sum-squared prediction error up to T timesteps in the future, using a variant of the backpropagation-through-time algorithm commonly used for recurrent neural networks [119].

When run online, my model will typically have some amount of past information, as I allow a short period where I optimize forces while a stiffness controller makes an initial inwards motion. Thus, simply initializing the latent state "cold" from some intial state and immediately penalizing prediction error does not match

well with the actual use of the network, and might in fact introduce overfitting by forcing the model to rely more heavily on short-term information. Instead, I train my model for a "warm" start. For some number of initial time-blocks B_{W}, I propagate latent state l, but do not predict or penalize system state $\hat{x}$, only doing so after this warm-up phase. I still back-propagate errors from future timesteps through the warm-up latent states as normal.

3.6 System Details

Learning System: I used the L-BFGS algorithm, shown to give strong results for deep learning methods [77], to optimize my model during learning. While larger network sizes gave slightly ($\sim$10%) less error, I found that setting $N_{lh} = 50$, $N_l = 50$, and $N_{oh} = 100$ was a good tradeoff between accuracy and computational performance. I found that block size $B = 10$, giving blocks of 0.1s, gave the best performance. When implemented on the GPU in MATLAB, all phases of my learning algorithm took roughly 30 minutes to optimize.

MPC Cost Function: In order to perform MPC, I need to define a cost function $C(X, U)$ for my task. For food cutting, I design a cost function with two main components, with β defining the weighting between them:

$$C(X, U) = C_{\mathrm{cut}}(X) + \beta C_{\mathrm{saw}}(X) \tag{3.15}$$

The first, C_{cut}, drives the controller to move the knife downwards. It penalizes the height of the knife at each timestep, with an additional penalty at the final timestep allowing a tradeoff between immediate and eventual downwards motion:

$$C_{\mathrm{cut}}(X) = \sum_{k=t}^{t+T} P_z^{(k)} + \gamma P_z^{(t+T)} \tag{3.16}$$

The second term, C_{saw}, keeps the tip of the knife inside some reasonable "sawing range" along the X axis, ensuring that it actually cuts through the food. Since any valid position is acceptable, this term will be zero inside some margins from this range, then become quadratic once it passes those margins. Taking P_x^* as the center point of the sawing range, d_{S} as the range, and λ as the margin, I define this term as:

$$C_{\text{saw}}(X) = \sum_{k=t}^{t+T} \left(\max\left\{0, |P_x^{(k)} - P_x^*| - d_{\text{S}} + \lambda\right\}\right)^2 \tag{3.17}$$

I also include terms performing first- and second-order smoothing and L2 regularization on the control forces.

Data Pre-Processing: In order to allow my learning algorithm to learn a better model for my data, I perform light pre-processing. I represent position features for each block relative to the last position in the previous block – e.g. if the previous block ended with an X-position of 0.4 m and the current block started with an X-position of 0.5 m, the first X-position feature for the current block would be 0.1 m. This representation avoids overfitting to absolute positions, while still representing relative motions and allowing me to easily reconstruct an absolute-position trajectory. Since I want to capture absolute, not relative, input forces, I do not offset them in this way.

The only whitening I perform on these features is to scale them so all features for a particular channel (e.g. X-positions, Z-forces, etc.) have unit standard deviation. I scale per-channel – applying the same scaling to all B features for a particular channel – rather than per-feature in order to preserve relative values within a channel. For similar reasons, I do not shift values e.g. to set the mean to zero as is common in other whitening approaches.

One advantage to my whitening approach is that it allows me to transfer this scaling to the input-layer weights during inference. For example, if I applied a scaling factor of 0.1 to X-position inputs during learning, I can simply scale the weights to the X-position used during inference by 0.1 and use un-whitened X-position values (still offset as above.) This saves computation time by avoiding performing scaling on new input features.

3.7 Real-time Robotic DeepMPC System

Robotic Platform: For both data collection and online evaluation of my algorithms, I used a PR2 robot. The PR2 has two 7-DoF manipulators with parallel-plate grippers, and a reach of roughly 1m. For safety reasons, I limit the forces applied by PR2's arms to 30N along each axis, which was sufficient to cut every material tested. PR2 natively runs the Robot Operating System (ROS) [118]. Its arm controllers recieve robot state information in the form of joint angles and must publish desired motor torques at a hard real-time rate of 1KHz.

Online Model-Predictive Control System: The main challenge in designing a real-time model-predictive controller for this architecture lies in allowing prediction and optimization to run continuously to ensure optimality of controls, while providing the model with the most recent state information and performing control at the required real-time rate. As shown in Fig. 3.6, I solve this by separating my online system into two processes (ROS nodes), one performing continuous optimization, and the other real-time control. These processes use a shared memory space for high-rate inter-process communication. This approach is modular and flexible - the optimization process is generic to the robot involved (given an appro-

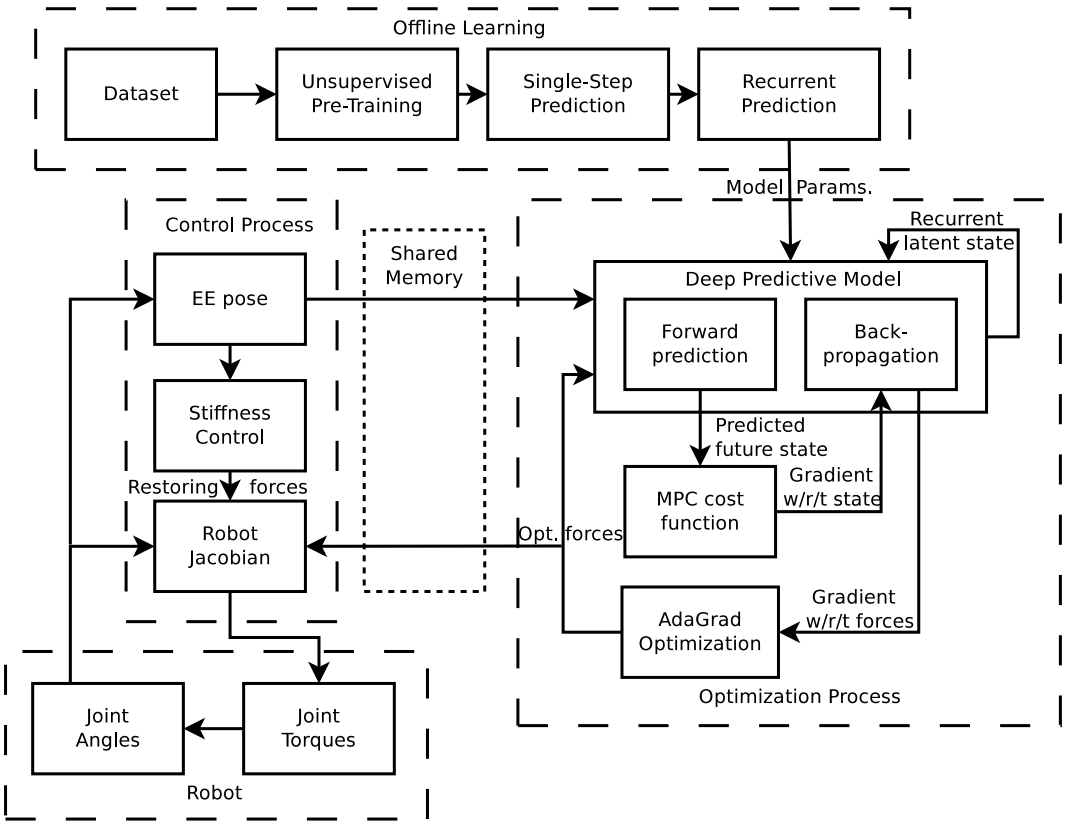


Figure 3.6: **Online system:** Block diagram of my DeepMPC system. Parameters learned using my three-stage deep learning algorithm are loaded by the optimization process, which then continually predicts future states and updates future controls based on these predictions. The control process takes state information from the robot, transmits it to the optimization process, and transmits controls optimized by that process to the robot.

priate model), while the control process is robot-specific, but generic to the task at hand. In fact, models for the optimization process do not even need to be learned locally, but could be shared using an online platform [130].

The **control** process is designed to perform minimal computation so that it can be called at a rate of 1KHz. It recieves robot state information in the form of joint angles, and control information from the optimization process as end-effector forces. It performs forward kinematics to determine end-effector pose, transmits it to the optimization process, and uses it to determine restoring forces for axes not

controlled by MPC. It translates the combination of these forces and those recieved from MPC to a set of joint torques sent to the arm. All operations performed by the control process are at most quadratic in terms of the number of degrees of freedom of the arm, allowing each call to run in roughly 0.1 ms on PR2.

The **optimization** process runs as a continuous loop. When started, it loads model parameters (network weights) from disk. Cost function parameters are loaded from a ROS parameter server, allowing them to be changed online. The optimization loop first uses past robot states (received from the control process) and control inputs along with past latent state and the future forces being optimized to predict future state using my model. It then uses this state to compute the gradients of the MPC cost function and back-propagates these through my model, yielding gradients with respect to the future forces. It optimizes these forces using a variant of the AdaGrad algorithm [34], a form of gradient descent in which gradient contributions are scaled by the L2 norm of past gradients, chosen because it is efficient in terms of function evaluations while avoiding scaling issues. This process is implemented using the Eigen matrix library [35], allowing the optimization loop to run at a rate of over 1.2kHz.

3.8 Prediction Experiments

In order to evaluate my model learning approach as compared to other state-of-the-art methods, I performed experiments evaluating prediction accuracy on my extensive dataset of 1488 material cuts. In the next section, I will also evaluate my algorithms on a real-world PR2 robot.

Dataset: My material interaction dataset contains 1488 examples of robotic food-

cutting for 20 different materials (Fig. 3.2). I collected data from three different settings. First, a fixed-parameter setting in which trajectories as shown in the leftmost two columns of Fig. 3.3 were used with a stiffness controller. Second, for 8 of the 20 materials in question, data was collected while a human tuned a stiffness controller to improve cutting rate. This data was not collected for all materials to avoid giving the learning algorithm and controller near-optimal cases for all materials. Third, a randomized setting where most parameters of the controller, including cutting and sawing rate and stiffnesses, but excluding sawing range (still fixed at 4cm) were randomized for each stroke of the knife. This helped to obtain data spanning the entire range of possible interactions.

Setting: In order to test my model, I examine its predictive accuracy compared to several other approaches. Each model was given 0.7s worth of past trajectory information (forces and known poses) and 0.5s of future forces and then asked to predict the future end-effector trajectory. For this experiment, I used 70% of my data for training, 10% for validation, and 20% for testing, sampling to keep each set class-balanced.

Baselines: For prediction, I include a linear state-space model, an ARMAX model which also weights a range of past states, and a K-Nearest Neighbors (K-NN) model (5-NN gave the best results) as baseline algorithms. I also include a GMM-linear model [81], and a Gaussian process (GP) model [30], trained using the GPML package [115]. Additionally, I compare to standard recurrent and non-recurrent two-layer deep networks, and versions of my approach without recurrence and without pre-training (randomly initializing weights, then training for recurrent prediction).

Results: Fig. 3.7 shows performance for each model as mean L2 distance from

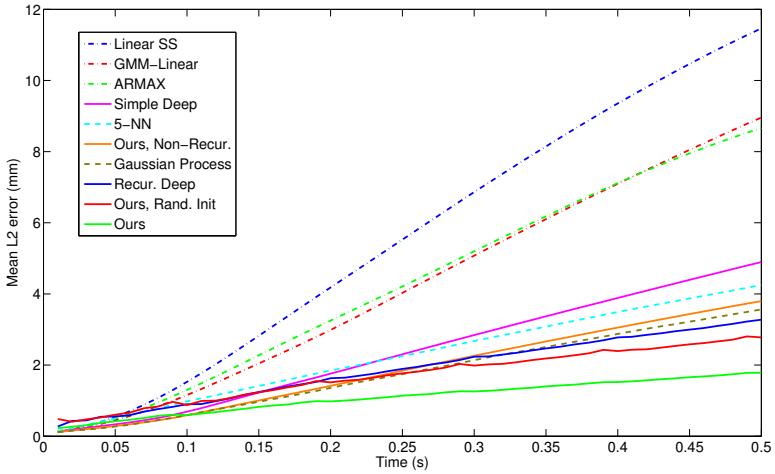


Figure 3.7: **Prediction error:** Mean L2 distance (in mm) from predicted to ground-truth trajectory from 0.01s to 0.5s in the future. While most models give similar performance up to 0.1s, models with linear components give very weak long-term results. Non-parametric methods give better results, but are hampered by expensive inference which scales poorly. Recurrent deep networks give the best results, with my approach outperforming all others after 0.1s, reducing error at 0.5s by 46% as compared to the baseline recurrent network

predicted to ground truth trajectory vs. prediction time in the future. Temporally-local (piecewise-) linear methods (linear SS, GMM-linear, and ARMAX) gave weak performance for this problem, each yielding an average error of over 8mm at 0.5s. This shows, as expected, that linear models are a poor fit for my highly non-linear problem.

Instance-based learning methods – K-NN and Gaussian processes – gave better performance, at an average of 4.25mm and 3.56mm, respectively. Both outperformed the baseline two-layer non-recurrent deep network, which gave 4.90mm error. The GP gave the best performance of any temporally-local model, although this came at the cost of extreme inference time, taking an average of 3361s (56 minutes) to predict 0.5s into the future, $1.18 \text{x} 10^6$ times slower than my algorithm, whose MATLAB implementation took only 3.1ms.

Table 3.1: **Confidence at 0.5s:** Mean L2 error and 95% confidence interval at prediction time of 0.5s (all in mm)

Algorithm	Mean	95% Conf.	
		Min.	Max.
Linear SS	11.46	0.89	38.58
GMM-Linear	8.96	0.58	31.80
ARMAX	8.66	0.79	31.36
Simple Deep	4.90	0.52	18.31
5-NN	4.25	0.22	19.24
Mine, Non-Recur.	3.80	0.35	15.03
Gaussian Process	3.56	0.27	14.85
Recur. Deep	3.27	0.47	12.14
Mine, Rand. Init	2.78	0.33	10.41
Mine	**1.78**	**0.16**	**7.55**

The relatively unimpressive performance of a standard two-layer deep network for this problem underscores the need for task-appropriate architectures and learning algorithms. Including conditional structures, as in the non-recurrent version of my model and temporal recurrence reduced this error to 3.80mm and 3.27mm, respectively. Even when randomly initialized, my model outperformed the baseline recurrent network, giving only 2.78mm error, showing the importance of using an appropriate architecture. Using my learning algorithm further reduced error to 1.78mm, 36% less than the randomly-initialized version and 46% less than the baseline recurrent model, demonstrating the need for careful pre-training.

For online model-predictive control, particularly when dealing with real-world variety, worst-case performance is very important, since I need to ensure that my model will perform well under any conditions. As shown in Table 3.1, my approach also gave a tighter and lower 95% confidence interval of prediction error, from 0.16-7.55mm at 0.5s, a width of 7.39mm, compared to the baseline recurrent net's interval of 0.47-12.14mm, a width of 11.67mm, and the GP's interval of 0.27-

14.85mm, a width of 14.58mm. In fact, my method was the only approach whose entire 95% confidence interval lay under 1cm of error.

3.9 Robotic Experiments

To evaluate my algorithm's performance for real-world robotic control, I performed an extensive series of over 450 experiments on my PR2 robot using the system described in Sec. 3.7.

Setting: In these experiments, the robot's goal was to make a linear cut between two given points. I selected a subset of 10 of the 20 materials in my dataset for these experiments, aiming to span the range of variation in material properties. I evaluated these experiments in terms of cutting rate, i.e. the vertical distance traveled by the knife divided by time taken.

Baselines: For control, I validate my algorithm against several other control methods with varying levels of class information. First, I compare to a class-generic stiffness controller, using the same controller for all classes. This controller was tuned to give a good (>90%) rate of success for all classes, meaning that it is much slower than necessary for easy-to-cut classes such as tofu. I also validate against controllers tuned separately for each of the test classes, using the same criteria as above, showing the maximum cutting rate that can be expected from fixed-trajectory stiffness control.

As a middleground, I compare to an algorithm similar to that of Gemici and Saxena [42], where a set of class-specific material properties are mapped to learned haptic categories. I found that learning five such categories assigned all data for

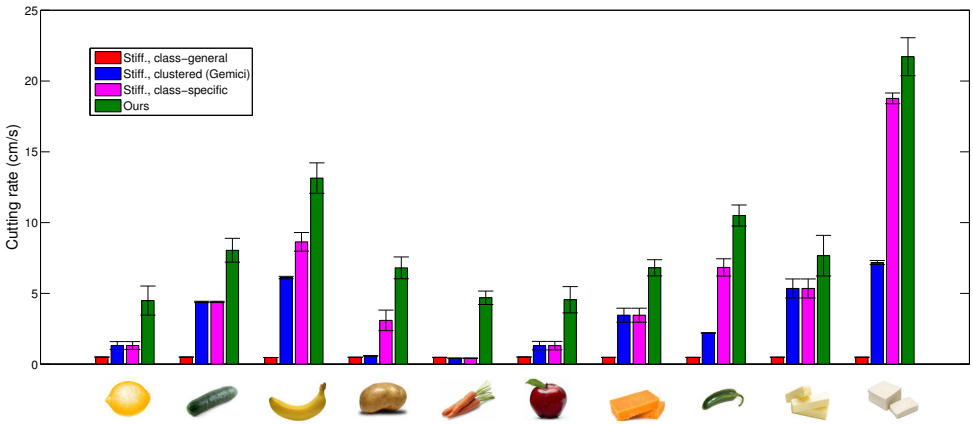


Figure 3.8: **Robotic experiment results:** Mean cutting rates, with bars showing normalized standard deviation, for ten diverse materials. Red bar uses the same controller for all materials, blue bar uses the same for each cluster given by [42], purple uses a tuned stiffness controller for each, and green is my online MPC method. My algorithm consistently gives higher mean rates, making statistically significant improvements for all materials except butter and tofu. Particularly significant improvements are seen for tough, varying materials such as carrots and potatoes.

each class to exactly one cluster, and thus used the same controller for all classes assigned to each cluster. In cases where this controller was the same as used for the class-tuned case, I used the same results for both.

Results: Figure 3.8 shows the results of my series of over 450 robotic experiments. For all materials except butter and tofu, a paired t-test showed that my DeepMPC controller made a statistically significant improvement in cutting rate with 95% confidence. This makes sense as butter and tofu are relatively soft and easy-to-cut materials. However, for the four materials for which stiffness control gave the weakest results – lemons, potatoes, carrots, and apples – my algorithm more than tripled the mean cutting rate, from 1.5 cm/s to 5.1 cm/s.

One major advantage my approach has over the others tested is that it treats material properties and classes as latent and continuous-valued, rather than supervised and discrete. For intra-class variations which affect dynamics, such as

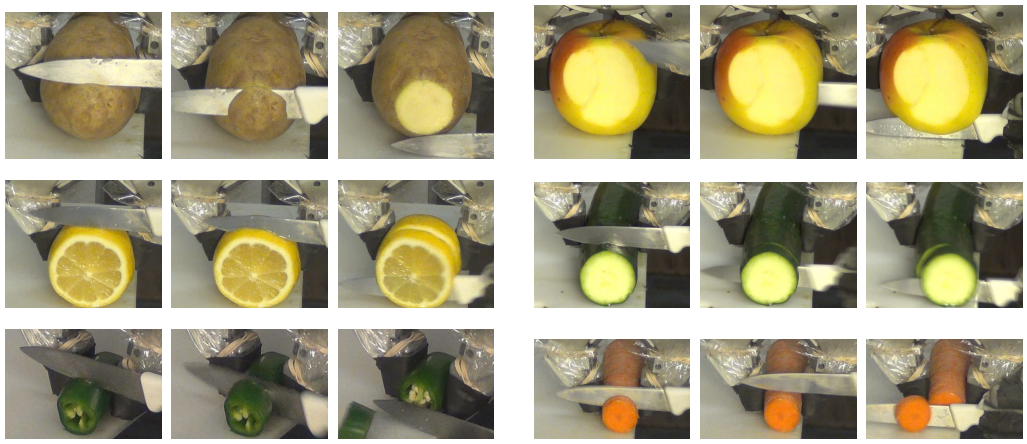


Figure 3.9: **Cutting food:** Time-series of my PR2 robot using my DeepMPC controller to cut several of the food items in my dataset. My algorithm is able to adapt its strategy for different materials. Note in particular that it picks up the knife slightly, then chops back down when cutting the carrot, and uses more "sawing" motion on tougher materials. Video of these experiments is available at `http://deepmpc.cs.cornell.edu`

different varieties of apples or cheeses, different radii of carrots or potatoes, or varying material temperature, even the class-specific stiffness controllers were typically limited by the hardest-to-cut variation. However, my approach's latent material properties allowed it to adapt to these, significantly increasing cutting rates. This was particularly evident for carrots, whose thickness causes huge variations in dynamics. While all approaches were tested on both thick and thin sections of carrot, only mine was able to properly adapt, slicing easily through thin sections and more carefully through thicker ones, increasing mean cutting rate from 0.4 cm/s to 4.7 cm/s. Similar results were observed for potatoes, increasing mean rate from 3.1 cm/s to 6.8 cm/s.

Another advantage of my approach is its ability to respond to time-dependent changes in dynamics, thanks to the time-varying nature of my latent features and the online adaptation performed by MPC. Such changes occur to some degree as the knife enters and becomes enclosed by most materials, particularly in irregular shapes such as potatoes where the degree of enclosure varies throughout the cut.

They are even more apparent in non-uniform materials, such as lemons, with variation between the skin and flesh, and apples, which are much denser and harder to cut closer to the core. Again, stiffness control was limited by the toughest of these dynamics, while my approach was able to adapt, typically performing more sawing for more difficult regions, and quickly moving downwards for easier ones. This allowed my controller to increase the mean cutting rate for lemons from 1.3 cm/s to 4.5 cm/s, and for apples from 1.4 cm/s to 4.6 cm/s.

Optimizing its trajectory online also enabled my DeepMPC controller to exhibit a much more diverse range of behaviors. Most tuned stiffness controllers were forced to make use of high-amplitude sawing to ensure continuous motion. However, my controller was able to use more aggressive cutting strategies, typically executing smooth slicing motions until it found its progress impeded. It then used a variety of techniques to break static friction and continue motion, including high-amplitude sawing, low-amplitude "wiggle" motions, and reducing and re-applying vertical pressure, even to the point of picking up the knife slightly in some cases. The latter behavior, in particular, underscores the strength of predictive control, as it trades off short-term losses for long-term gains. Stiffness controllers sometimes became stuck in tough materials such as thick potatoes and carrots and the cores of apples, and remained so because downwards force grew as vertical error increased. My controller, however, was able to detect and break such cases using these techniques.

Some examples of the diverse behaviors of my DeepMPC controller can be seen in Fig. 3.9 and in the video at `http://deepmpc.cs.cornell.edu`.

3.9.1 Handling Variety

In addition to my main experiments, presented above, I performed a series of additional experiments to test the adaptive abilities of my controller. For both these experiments, I chose to use potatoes, as they represent a challenging, dense material which responds in interesting ways to changes in temperature and tools (as detailed below.)

Varying Temperature: While the temperature of a material is not detectable by most robotic platforms (with some notable exceptions [11]), it can have a huge effect on the cutting dynamics of that material. I performed a series of experiments to validate the robustness of my algorithm to this variation. For these experiments, I prepared several "cold" potatoes by placing them in a freezer for 30 minutes, and several "warm" potatoes by microwaving them for five minutes. Both of these operations significantly altered the texture of the potato.

I did not tune or alter any controllers to reflect these new cases. As a baseline, I compare to the class-specific stiffness controller tuned for potatoes described above. For my approach, I used the same learned controller as used in all other robotic experiments, which has no experience with either warm or cold potatoes, only room-temperature. These are the same controllers presented as the purple and green bars, respectively, in Fig. 3.8, which gave cutting rates of 3.1 cm/s and 6.8 cm/s for room temperature potatoes.

The warm potatoes were much softer and easier to cut than at room temperature. This allowed the stiffness controller to track its input trajectory almost exactly, increasing cutting rate slightly to 3.5 cm/s. My approach, however, was able to properly adapt to these new conditions, almost doubling its cutting rate

Figure 3.10: **Different tools:** Different knives used to test my algorithm. From left to right, the paring knife used to collect data and train the algorithm, a shorter, sharper paring knife, a long kitchen knife, a wedge-shaped chef's knife, and a serrated steak knife. In all cases except the serrated knife, my algorithm, trained only with the paring knife on the left, was able to maintain comparable cutting rates.

to 12.0 cm/s.

The cold potatoes were significantly more difficult to cut. In over half of my trials with cold potatoes, the baseline controller became stuck, with downwards progress halting after some point. For purposes of rate computation, I considered the controller to still be cutting until it reached the end of its input trajectory, leading to an average cutting rate of only 1.8 cm/s. My controller, once again, was able to properly adapt to this new case. While its forward progress was sometimes temporarily halted, it was able to perform motions to break out and continue cutting, allowing it to achieve a rate of 3.4 cm/s, almost double that of the baseline stiffness controller.

Varying Tools: I also tested my controller, learned using the paring knife shown in Fig. 3.4, with a series of other knives, as shown in Fig. 3.10. These included a

much sharper and slightly shorter paring knife, a longer kitchen knife, a wedge-shaped chef's knife, and a serrated steak knife. My controller was able to adapt to the first three knives, giving cutting rates similar to the results in Fig. 3.8. The sharp paring knife increased the rate slightly, to 7.8 cm/s, while the long and wedge knives both lead to slightly decreased rates of 5.5 and 5.6 cm/s, respectively. This makes sense, as the sharp paring knife was very similar to the training knife, only sharper, wheras the long and wedge-shaped knives were significantly different, leading to different cutting dynamics, particularly in the case of the wedge-shaped knife. It is notable that my algorithm experienced only a slight (roughly 17%) decrease in performance for such a significant change in tools.

My algorithm, trained using a straight-edged paring knife, was not able to use the serrated steak knife to cut a potato. This makes sense for two reasons:

First, a serrated knife is not, in general, a good tool for cutting dense vegetables like potatoes. Typically, such knives are only used for meat, where the main mode of cutting is splitting muscle fibers, or soft, skinned vegetables such as tomatoes, where the serration helps to cut through the skin without smashing the vegetable, which offers minimal resistance. For dense vegetables, a sharp, straight edge is necessary to split the vegetable, and even a skilled human user would have a hard time using a serrated knife.

Second, the dynamics of cutting using a serrated knife are very different – while a straight-edged knife acts primarily by splitting the vegetable apart, a serrated knife acts by "eroding" the food item with its serrations. Food will respond very differently to the same motion executed by a serrated knife, so such a knife requires a significantly different dynamics model and cutting strategy. In particular, straight downwards motions are ineffective, requiring instead more "sawing"

and less downwards force to allow the serrations to act. Thus, experience with a straight-edge knife should not be expected to transfer to a serrated one. However, given new training data, my algorithm could also learn to use such a knife.

In both these experiments, my algorithm was able to successfully handle variations that it was not explicitly trained for. While varying temperature and tools both significantly alter cutting dynamics, in all cases but the serrated knife, my algorithm was able to adapt to these and maintain comparable cutting rates, even improving them in some cases.

3.10 Conclusion

In this work, I presented DeepMPC, a novel approach to model-predictive control for complex non-linear dynamics which might vary both with environment properties and with time while acting. Instead of hand-designing predictive dynamics models, which is extremely difficult and time-consuming for such tasks, my approach uses a new deep architecture and learning algorithm to accurately model even such complicated dynamics. In experiments on my large-scale dataset of 1488 material cuts over 20 diverse materials, I showed that my approach improves accuracy by 46% as compared to a standard recurrent deep network. In a series of over 450 real-world robotic experiments for the challenging problem of robotic food-cutting, I showed that my algorithm produced significant improvements in cutting rate for all but the easiest-to-cut materials, and over tripled average cutting rates for the most difficult ones.

Here, I presented a system which displays adaptability while learning a good latent representation for complex tasks. While food-cutting and many other ma-

nipulation tasks lend themselves well to intuitive cost functions, for others I could envision also learning the cost function used for MPC from data. In future work, this system could also be extended to handle multimodal information, e.g. incorporating haptic, visual, audio, or other data to enhance manipulation. While it would be extremely difficult to design features which properly integrate all this information, deep learning allows me to learn these features directly from data, and my system would allow me to integrate them into real-time model-predictive control.

Algorithm 1 Recurrent Prediction and Cost Gradients for MPC

Input:

Previous and current dynamic responses $v^{(b-1)}$, $v^{(b)}$

Current latent state $l^{(b)}$

Future control inputs $U_{t+1:t+T}$

Output:

Predicted future state $\hat{X}_{b+1:b+T}$

Cost function value $C(\hat{X}_{b+1:b+T}, U_{b+1:b+T})$

Cost gradients $\partial C(\hat{X}_{b+1:b+T}, U_{b+1:b+T})/\partial U_{b+1:b+T}$

Define $C(\hat{x}^{(i)}, U)$ as the direct contribution of $\hat{x}^{(i)}$ to $C(\hat{X}_{t+1:t+T}, U_{t+1:t+T})$ (ignoring recurrent effects.)
Define $C(U)$ as the direct contribution of U to the cost function (e.g. via smoothing/regularization terms)

Forward prediction:

for k = 1:T

Compute $l^{(b+k)}$ from eq. (3.9)

Compute $\hat{x}^{(b+k)}$ from eq. (3.6)

end

Compute $C(\hat{X}_{t+1:t+T}, U_{t+1:t+T})$

Back-propagating gradients:

`dXBack1` = 0; `dXBack2` = 0; `dUBack1` = 0; `dUBack2` = 0; `dLBack1` = 0;

for k = B:1

Compute current cost function gradient, back-prop to u

`dXCur` = $\partial C(\hat{x}^{(b+k)}, U)/\partial \hat{x}^{(b+k)}$

`dXCur` = `dXCur` + `dXBack1`;

$\partial C(\hat{X}, U)/\partial u^{(b+k)}$ = `dXCur` * $\partial \hat{x}^{(b+k)}/\partial u^{(b+k)}$
+ $\partial C(U)/\partial u^{(b+k)}$ + `dUBack1`

Back-propagate gradients to previous timesteps

`dXBack1` = `dXBack2` + `dXCur` * $\partial \hat{x}^{(b+k)}/\partial \hat{x}^{(b+k-1)}$
+ `dLBack1` * $\partial l^{(b+k)}/\partial \hat{x}^{(b+k-1)}$

`dUBack1` = `dUBack2` + `dXCur` * $\partial \hat{x}^{(b+k)}/\partial u^{(b+k-1)}$
+ `dLBack1` * $\partial l^{(b+k)}/\partial \hat{u}^{(b+k-1)}$

`dXBack2` = `dXCur` * $\partial \hat{x}^{(b+k)}/\partial \hat{x}^{(b+k-2)}$
+ `dLBack1` * $\partial l^{(b+k)}/\partial \hat{x}^{(b+k-2)}$

`dUBack2` = `dXCur` * $\partial \hat{x}^{(b+k)}/\partial u^{(b+k-2)}$
+ `dLBack1` * $\partial l^{(b+k)}/\partial \hat{u}^{(b+k-2)}$

`dLBack1` = `dLBack1` * $\partial l^{(b+k)}/\partial l^{(b+k-1)}$
+ `dXCur` * $\partial \hat{x}^{(b+k)}/\partial l^{(b+k-1)}$

end

CHAPTER 4

LOW-POWER PARALLEL ALGORITHMS FOR SINGLE IMAGE BASED OBSTACLE AVOIDANCE IN AERIAL ROBOTS

4.1 Introduction

Perceiving obstacles is extremely important for an aerial robot in order to avoid collisions. Methods based on stereo vision [101, 52] are fundamentally limited by the finite baseline between the stereo pairs [97], and fail in textureless regions and in presence of specular reflections [10]. Active range-finding devices (e.g., [150, 93]) are either designed for indoor low-light environments (e.g., the Kinect [3]), or are too heavy for aerial applications. More importantly, they demand more onboard power, which is at a premium for aerial vehicles.

In this work, I use a single monocular camera for obstacle perception. Recent works [129, 94, 123, 127, 125, 5] have shown that it is possible to obtain depth from a single monocular image. Multiple frames can also be used in combination to determine depth, but this approach does not work well on an aerial robot due to camera disturbances from robot motion and vibrations. Here, I present an algorithm that takes a single image as input and classifies each region in the image as obstacle or not. I will define an obstacle as an object which the robot could not safely pass through. This approach is very attractive for aerial robots because cameras are small and draw little power.

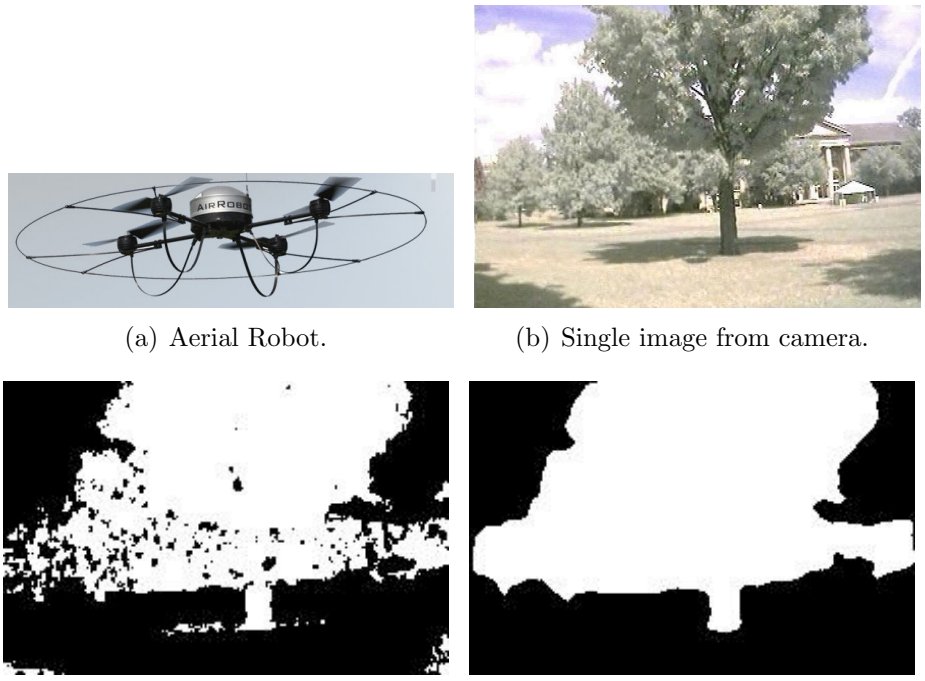
My second key contribution is to formulate a Markov Random Field classifica-

This work was originally presented as a conference paper at the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) 2012. It is joint work with Mevlana Gemici and Ashutosh Saxena.

tion model and design fast inference algorithms for estimating obstacles in a new low-power parallel hardware that implements a leaky integrate and fire neuron architecture. One implementation of such hardware [92] consumes only 45 pico-Joules per spike (see Section 4.3 for an overview). The combination of using a camera (miniature cameras that consume extremely low power are available) and this hardware could allow miniature aerial robots to successfully fly amidst obstacles even in unknown environments. Inference in my MRF is solved using Belief Propagation (BP). While performing belief propagation in a traditional computer is expensive, my design allows the low-power hardware to do so natively. Each logical clock cycle performs a full parallel update of BP. I obtain good obstacle estimates once the BP network has converged (see Figure 4.1). I then use the estimated obstacle map to select a evasive manuever for avoiding the obstacle.

In detail, I represent the energy function of the MRF over the neuron architecture, and use spikes to propagate beliefs during inference. My MRF uses the logistic function to model the local dependence of visual features to the obstacle label, where each spatial region of the image is represented using a different set of neurons—this allows parallel computation. Furthermore, I use different parameters for different spatial regions of the image [83] for improved performance. Finally, I induce sparsity in the parameters of the model to satisfy constraints on the number of parameters allowed by my hardware.

I performed extensive experiments in a variety of environments, containing obstacles such as trees, fences, and poles. In learning experiments, I obtained an average precision and recall of 81.9% and 93.6% respectively. In 53 outdoor robotic experiments, my algorithm was able to successfully perceive obstacles in every case, and avoid them in 51 cases. The two failures were due to communication delays



(a) Aerial Robot. (b) Single image from camera.

(c) Initial inferred obstacle map. (d) Inferred obstacle map.

Figure 4.1: **Avoiding obstacles:** I use low-power parallel hardware to compute an obstacle map, given a single image from the camera onboard the aerial robot. I then use these results to select an evasive maneuver.

and robot drift. Some of these experiments involved avoiding a series of obstacles of multiple types.

4.2 Related Work

In aerial robotics, most works which perform obstacle avoidance either make strong assumptions on precise knowledge of 3D location of obstacles [90], or use sensors that are not onboard, such as GPS (together with known obstacle map). Other work such as [48] focuses on mapping obstacles from overhead images. For a small aerial robot to fly autonomously in a real environment full of obstacles, these techniques do not directly apply.

Navigation by labeling obstacles in images has been used for several ground robots. For example, Ghosh and Mulligan [43] use a ground segmentation approach for navigation, while Nabbe and Hebert [102] use ground-vertical segmentation for extending the path planning horizon for ground robots. Work such as [67, 135] employs learning algorithms to determine terrain traversability for ground vehicles. Michels, Saxena and Ng [94] and Plagemann et. al. [111] attempt to determine range directly from monocular images. However, these works use only a local feature based classifier for navigating a ground vehicle. An aerial robot is typically severely constrained by onboard power, and I present methods that allow even a complex inference method such as BP to be efficiently computed in low-powered hardware.

There are other works that consider single monocular image based obstacle avoidance for aerial robots. McGee et. al. [89] use sky segmentation for detecting obstacles, but apply only a local classifier. Soundararaj, Sujeeth and Saxena [137] and Courbon et. al. [29] use vision techniques to navigate aerial robots, but are limited to known indoor environments. Bills, Chen and Saxena [13] and Zingg et. al [152] perform similar work to unknown environments, but still handle only a few known types of indoor environment. On the other hand, I consider general outdoor environments, employing learning algorithms which allow my approach to be easily adapted to new obstacle types and integrate non-local information to enhance classification.

Vision algorithms have implemented in neural architectures or embedded systems, such as [69, 4, 64]. Other works [99, 57] used spiking neurons for basic obstacle detection and navigation. However, these approaches do not generalize to real-world outdoor cases.

4.3 Neuromorphic Hardware

My goal is to develop algorithms for obstacle mapping that can be implemented in low-power parallel hardware. In particular, I use a neuromorphic hardware platform that comprises a network of linear-leak integrate and fire (LLIF) artificial neurons as in [92]. The LLIF neurons represent an extremely versatile high-level primitive which couples memory and processing. More importantly, this architecture uses extremely low power, as discussed below.

Each neuron integrates the weighted synaptic inputs from other neurons and fires if the integrated value exceeds a preset threshold. More formally, each neuron has some integer-valued internal potential Z and binary-valued spiking output S. For i^{th} neuron, S and Z update as:

$$\begin{aligned} Z_t^{i+} &= Z_t^i + \sum_{j \in \mathcal{N}(i)} w_{ij} Y_t^j - \lambda_i \\ Z_{t+1}^i &= \mathbf{1}\{Z_t^{i+} < \alpha_i\} Z_t^{i+} \\ S_{t+1}^i &= \mathbf{1}\{Z_t^{i+} \geq \alpha_i\} \end{aligned} \tag{4.1}$$

where $\mathcal{N}(i)$ are the neighbors of neuron i, $w_{i,j}$ indicates the synaptic weight from neuron j to neuron i, α indicates neuron threshold, and λ is a constant decay. $\mathbf{1}\{...\}$ is the indicator function, which takes the value one if its argument is true and zero otherwise.

Since these are spiking neurons, one major restriction is that the inputs and outputs be binary-valued. I address this by using representations where the spike count over a particular time window is proportional to the value being represented. Since each neuron can integrate over time, this is still a useful representation. The expected spike rate given the input x is simply a ramp function with limits, as

follows:

$$g(x,\alpha) = \begin{cases} 0 & \text{if } x \leq 0 \\ x/\alpha & \text{if } 0 < x < \alpha_i \\ 1 & \text{if } x \geq \alpha_i \end{cases} \tag{4.2}$$

Cardinality constraints in hardware. To allow for a more compact design with lower power consumption, hardware such as that in [92] typically imposes a constraint on the cardinality of the weights w. That is to say, each neuron's weights may take at most k unique nonzero values.

Power consumption in hardware. In a well-designed hardware platform such as [92], power consumption will be proportional to the number of spikes and the density of connections. In particular, the hardware in [92] takes only 45 pico-Joules (pJ) per spike, and has very low quiescent power draw in their absence.

4.4 Obstacle Classification

The primary goal of my approach is to produce an obstacle map of sufficient quality for obstacle avoidance. This is a challenging problem, as outdoor environments are perceptually complex, with variations in obstacle appearance, lighting, background appearance, and other factors. My model will define obstacles as objects which project upwards from the ground and thus present a navigational challenge to the robot. I will use the labels it produces to select an evasive maneuver for the perceived obstacles.

My classification model is a Markov Random Field (MRF) model (e.g., see [72]), where I use an Ising pairwise potential for modeling dependencies between

neighboring image regions. More formally, an MRF is an undirected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where I represent each labeled location in the image as a vertex $\mathcal{V}$, and edges $\mathcal{E}$ connecting neighboring image locations. Let $Y^i \in \{-1, +1\}$ represent the binary labels indicating presence or absence of an obstacle at the i^{th} location in the image, and X^i represent the input visual features at that location.

I model the joint conditional likelihood of the labels given the features as:

$$P(Y|X,\theta) \propto \exp\left(-E(Y|X,\theta)\right) \tag{4.3}$$

where the $E(Y|X,\theta)$ is an energy function containing three terms:

$$E(Y|X,\theta) = -\sum_{i\in\mathcal{V}} Y^i A(X^i,\theta) - \sum_{(i,j)\in\mathcal{E}} w_{ij} Y^i Y^j + \beta||\theta||_1 \tag{4.4}$$

The first term uses a logistic model, with θ as its parameters, to model the dependence of the label on the local visual features. More formally, I model the association potential as $A(X,\theta) = 2 * \sigma(X^T\theta) - 1$, where $\sigma(x) = 1/(1+\exp(-x))$ is the sigmoid function. The second term prefers neighboring labels to be similar, with w indicating the relative importance of the two terms. Finally, β controls the strength of an L_1 regularization term on the local feature weights, which helps with the weight cardinality constraints in the hardware.

During learning, I will be given a set of labeled examples and my goal is to find the optimum value of the parameters. During inference, the system is given a new image and my goal is to find the optimal value of Y using the LLIF neuron architecture.

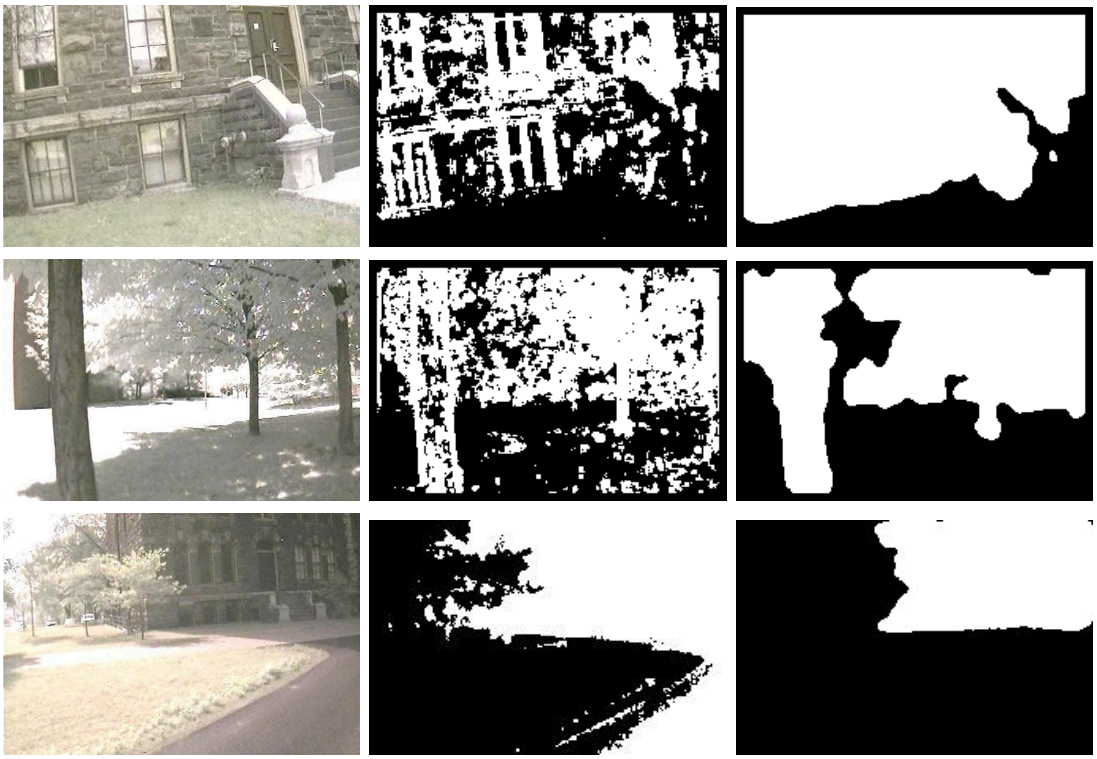


Figure 4.2: **Classification improvements from MRF:** Left: input images. Middle: initial classification. Right: classification with full MRF model with belief propagation. Initial classification results which would present problems for navigation, but are greatly improved by integrating non-local information using my MRF.

4.4.1 Learning

I manually set parameters w_{ij}, and learn parameters θ by maximizing the pseudo-conditional log-likelihood. The system is given M labeled ground-truth pairs as $\{(X_m, Y_m) : m = 1, \ldots, M\}$, and I learn θ^* as:

$$\theta^* = \arg\min_{\theta} \sum_{m=1}^{M} E(Y_m | X_m, \theta) \tag{4.5}$$

$$= \arg\min_{\theta} \sum_{m=1}^{M} \left(- \sum_{i \in \mathcal{V}} Y_m^i A(X_m^i, \theta) \right) + \beta ||\theta||_1 \tag{4.6}$$

Here, Y_m^i indicates Y^i from the m^{th} training example, and X_m^i is similar for input features. This sub-problem is convex, and is equivalent to solving logistic regression with an additional L_1 penalty term. I vary the β parameter until the number of

non-zero weights fits within what is allowed in the hardware.

4.4.2 Inference

The goal of inference is to find an optimal value for the label estimates Y, given features X and parameters θ:

$$Y^* = \arg\max_Y P(Y|X,\theta) \tag{4.7}$$

My inference algorithm is based on loopy belief propagation [109, 100]. In order to derive the update rule for node i, I assume that the values of the other nodes are known and compute the messages as follows:

$$\psi(X^i) = \sigma(\theta^T X^i) \tag{4.8}$$

$$\mu_i(X^i) = \psi(X^i) \prod_{(i,j)\in\mathcal{E}} \mu_j(X^j)^{w_{ij}} \tag{4.9}$$

$$P(Y^i = 1|X) = \mu_i(X^i) \tag{4.10}$$

My goal is to perform obstacle avoidance in new environments, and the product above may give zero probability of being an obstacle if any of terms is zero. This is not preferable, and I need to account for such cases. Following ideas of additive smoothing in statistics [145], I include an additive smoothing term in ψ and μ. This is a small constant factor ϵ added to each function. Denote the versions of these functions with additive smoothing as ψ_s and μ_s.

If I consider these update equations in log-space, they become sum of weighted terms, and thereby can be implemented in neuromorphic hardware (Eqn. 4.1). In such a case, the additive smoothing term becomes a constant lower bound on the

log probability terms. Thus, in log-space, I have:

$$\log \psi_s(X^i) = \max(\log \epsilon, \log \sigma(\theta^T X^i)) \tag{4.11}$$

$$\log \mu_{s,i}(X^i) = \max \big(\log \epsilon, \log \psi_s(X^i) + \sum_{(i,j)\in\mathcal{E}} w_{ij} \log \mu_{s,j}(X^j) \big) \tag{4.12}$$

To implement this in hardware, I will use one neuron each to represent $\psi_s(X^i)$ and $\mu_{s,i}(X^i)$ for each node. Each ψ unit will take input in spike-rate from local features, weighted as θ. Each μ unit will take input from the corresponding ψ unit and neighboring μ units, weighted as w.

The log-sigmoid ψ_s term can be approximated as a linear function which saturates at 0. With the lower bound from the additive smoothing function, this becomes a scaled and shifted version of Eqn. 4.2. I will refer to the version of $g(x, \alpha)$ adjusted to fit the log-sigmoid function as $g_\psi(x)$. $\log \mu_{s,i}(X^i)$ is also well approximated by a thresholded linear function, and can thus also be modeled by $g(x, \alpha)$, as $g_\mu(x)$. In both cases, α is fixed to some value which gives the best fit. The equations for my system are then:

$$\log \psi_s(X^i) = g_\psi(\theta^T X^i) - \log Z_\psi \tag{4.13}$$

$$\log \mu_{s,i}(X^i) = g_\mu(w_{ii}\psi_s(X^i) + \sum_{(i,j)\in\mathcal{E}} w_{ij}\mu_{s,j}(X^j)) - \log Z_\mu \tag{4.14}$$

Where the Z's are two separate constants, necessary to include here to preserve exact equality, but unnecessary to implement in hardware since relative values are preserved. Except for the error introduced by approximating the log-sigmoid function with $g_\psi(x)$ and discretization, this is exactly Eqn. 4.12. To infer optimal values for Y, I can simply threshold the g_μ terms once the network has converged.

4.4.3 Visual Features

My basic features are a standard set of texture filter responses, similar to those in [87]. They consist of oriented edge filters (in my case, Gabor filters) and center-surround filters (difference-of-Gaussian filters) as shown in Figure 4.3. I also include color information in the form of patch-averaged RGB values in my feature set. In addition to raw filter responses, I include the absolute value of these responses and the maximum and average of this absolute value over a local area.

I found it difficult to design features which fit the hardware constraints of [92], so the features given here do not. They do, however, rely only on simple local computations, and thus are amenable to efficient implementation in circuitry or parallel hardware. Only features given nonzero weights by some classifier would need to be implemented in this hardware, significantly reducing complexity.

Figure 4.3: **Filter set:** My filter set, which includes two scales of Gabor filters at six orientations and two scales of difference-of-Gaussian filters.

4.4.4 Spatially-varying models and multiple models

Often the statistics of the dependence of the obstacles on the visual features varies with their location in the image [83]. For example, while leaves on the trees are typically obstacles, they are not if they fall on the ground in the Fall season. Thus

my parameters θ should be different for different rows in the image. In order to do so, I divide the image vertically into three equally sized areas. This is equivalent to considering a separate θ_i for each $i \in \mathcal{V}$, and tying weights within regions. By dividing the classification as such, each region can have a different feature set. Tying the parameters of these different models helps to avoid over-fitting. Since my neuron architecture is distributed for different spatial regions in the image, each neuron gets the appropriate synaptic weights depending on which region in the image it is representing.

To detect multiple visually different obstacle classes, I train a model for each class and use the combined output of these models. In order to exercise maximum caution, I combine outputs using the logical OR of the results, ie if any classifier classifies a region as obstacle, it is considered to be an obstacle for purposes of navigation.

Since my algorithms run natively in parallel hardware, neither of these changes require a change to the architecture or cause an increase in runtime.

4.4.5 Tuning for Power Consumption

One major strength of the hardware described in [92] is that its power consumption can be estimated a priori from simulation with high accuracy. This is because power consumption in this hardware is determined largely by two factors: a) connection density and b) spike count. While a) is fixed for a particular local connection pattern, b) can change drastically depending on the relative weight between local classifier estimates and neighboring labels, as well as neuron threshold and decay. In particular, faster convergence of the MRF model in terms of hard-

ware timesteps can significantly reduce power consumption, as the model can be terminated earlier and thus generates fewer spikes. Therefore, in practice, I tune my models for a tradeoff between power and accuracy, aiming to produce results which produce fewer spikes while still yielding results which would be useful for obstacle avoidance.

In practice, I obtained good performance in terms of both accuracy and performance for the following parameter settings for each neuron:

- w_{ii}: set slightly below α (firing threshold).
- w_{ij}: set to a small local connectivity pattern (4- or 8-way), with uniform weights set to 10-20% of α.
- λ (decay): set to 5-10% of α.

4.5 Obstacle Avoidance Manuevers

The goal of my obstacle avoidance algorithm is to avoid obstacles in near to mid-range (e.g., about 2m to 10m). My motion selection algorithm uses a library of paired motions and visual-space masks. If less than some threshold's worth of pixels within a masked area are labeled as obstacles, the corresponding motion is considered to be safe. The effects of sweeping this threshold are shown in Figure 4.6. A fixed preference order is used to select motions in cases where more than one is found to be safe.

For vertical obstacles such as trees and poles, the motions, in order of preference, were diagonal-left, diagonal-right, and forwards. For horizontal obstacles such as fences, the motions were forwards or upwards.

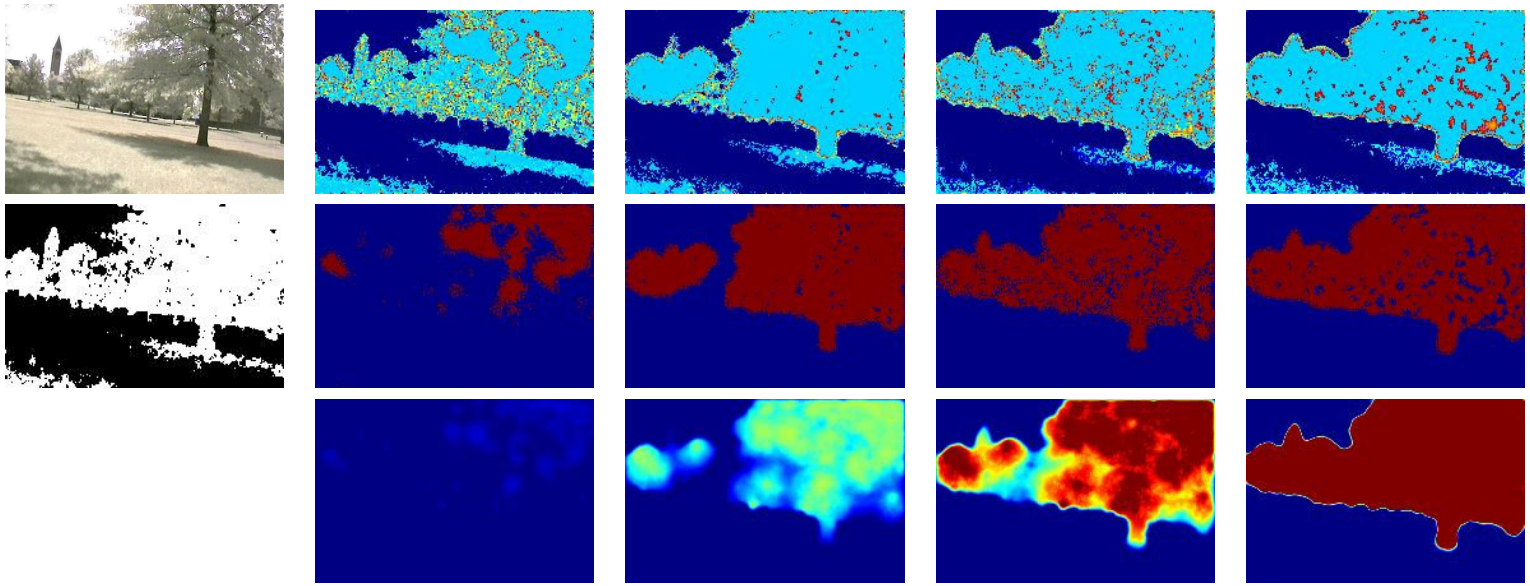


Figure 4.4: **BP in action:** Time series of belief propagation in action. Left: original image and baseline classification result. Top: internal node potential. Middle: spike locations. Bottom: spike counts. Time proceeds from left to right. Best viewed in color.

Table 4.1: **Obstacle classification results:** Classifier precision and recall, in percent, for four obstacle classes, and average across classes. Results presented for baseline local classifier and MRF model, with and without spatially varying model, and varying weight cardinalities.

Algorithm	Card.	Trees		Buildings		Fences		Poles		Average	
		Prec.	Rec.	Prec.	Rec.	Prec.	Rec.	Prec.	Rec.	Prec.	Rec.
Chance	-	50.7	52.0	70.1	65.3	28.9	21.5	5.4	4.4	29.3	26.4
Local association term only, no spatial-varying model	110	80.0	81.5	83.6	91.0	61.5	92.6	56.6	94.7	71.8	88.2
	8	78.6	77.2	86.9	89.4	68.3	90.0	62.2	92.6	74.8	86.0
	5	77.4	75.4	90.3	73.2	67.6	90.1	47.9	95.6	70.8	86.2
	3	77.2	75.3	88.7	78.2	66.8	90.7	47.7	95.4	69.7	86.4
Local association term only, spatial-varying model	110	75.9	85.5	91.2	86.1	80.3	92.1	54.0	86.7	75.4	87.6
	8	74.1	84.0	91.7	85.3	78.2	93.0	59.8	86.6	75.9	87.2
	5	72.8	81.3	91.0	85.6	77.4	93.2	56.3	86.8	74.4	85.6
	3	70.5	77.9	90.7	79.9	66.6	93.4	57.0	86.8	71.2	84.5
Full MRF, no spatialy varying model	110	79.8	94.8	89.0	94.2	82.1	87.4	80.8	91.7	82.2	92.3
	8	76.3	91.9	84.8	95.0	89.2	85.3	82.1	90.7	83.5	91.1
	5	76.0	92.0	90.3	89.6	84.1	87.9	80.6	90.2	81.7	91.5
	3	71.5	88.1	87.1	90.0	90.2	89.2	81.8	89.9	82.2	90.7
Full MRF, spatialy varying model	110	81.9	94.0	87.7	96.3	95.0	94.0	63.1	90.1	81.9	93.6
	8	78.9	92.7	87.6	96.2	92.2	93.9	65.3	89.1	81.0	93.0
	5	79.3	92.0	87.2	95.6	91.1	93.6	65.7	89.6	80.8	92.8
	3	74.4	90.1	86.9	95.8	90.3	92.5	64.2	88.5	79.0	91.7

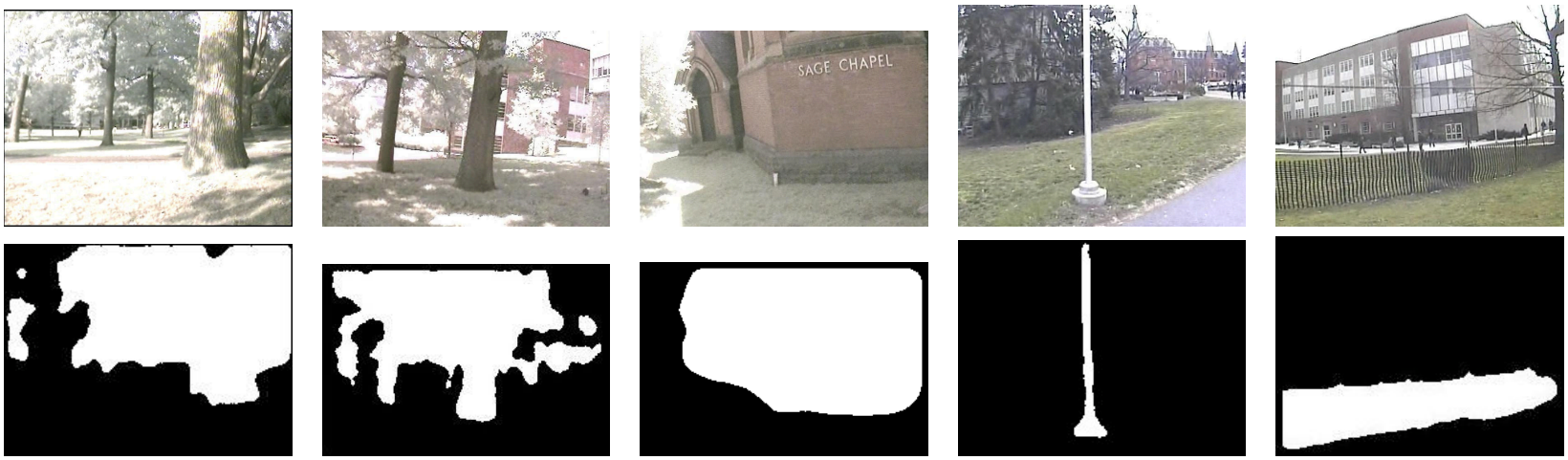


Figure 4.5: **Visual results:** Figure showing the results of my algorithm for a variety of obstacles. Top: Input image. Bottom: Inferred obstacle labels.

4.6 Experiments and Results

4.6.1 Offline Learning Experiments

Dataset. My dataset includes 120 images taken from various locations around the Cornell University campus using the onboard camera of my aerial robot. The environments include four types of obstacles: trees, buildings, light poles, and fences. The dataset includes 63 images of trees, 45 of buildings, 10 of poles and 10 of fences. I used 80% of the images for training and 20% for testing.

Belief Propagation. During inference, my BP system exhibits distinct phases of operation as seen in Figure 4.4. First, there is a warm-up phase where nodes with high values of classifier output build up energy. Eventually, some of these nodes' local potential exceeds their thresholds and they fire, spreading energy to neighboring nodes which may also fire in response. Spikes begin to propagate across the network, which finally reaches a steady state from which inferred classification labels are determined.

Results. Table 4.1 shows the performance of my algorithm. I present comparisons with different models. First, I consider a model with only the association term, i.e., equivalent to a local logistic classifier. Second, I also compare the effects of training several spatially varying models. I compare the effects of reducing the cardinality of local classifier weights as well. Similar baseline results were observed using SVM.

My results show that the spiking neuron based BP system has proven very effective in producing cleaner, more usable results for obstacle avoidance, as com-

pared to baseline results using the local logistic term only. Since my feature set contains edge and center-surround filter responses, local classifier estimates are generally stronger at obstacle edges. BP propagates these inwards, allowing the entire obstacle to be classified, causing increases in recall of as much as 17%. This propagation behavior can sometimes smooth edges of perceived obstacles slightly, as seen in many cases in Figure 4.5, causing a slight decrease in precision offset by gains elsewhere.

Most clear areas contain no strong local values and thus do not generate an initial spike, while true positive regions almost always do. This allows the system to avoid some false positive cases present in the baseline results. For obstacle avoidance purposes, recall is much more important than precision, and errors near an obstacle are less important than false positives in otherwise clear areas.

In order to evaluate the performance of my algorithm for obstacle avoidance purposes, I discretized the lower region of each test image into a 3x5 grid of cells, and considered a cell to be ground-truth occupied if it contained at least 10% ground-truth obstacles. I produced the curve shown in Figure 4.6 by sweeping thresholds for occupancy ratios. My algorithm outperforms the baseline in all cases. The most significant improvement is for high values of recall, which are necessary for safe obstacle avoidance. Results presented are for trees, results for other classes were similar.

Weight Cardinality Limitations. Reducing the number of features available to the classifier decreases performance on average, producing particularly significant decreases for varied obstacles such as trees. However, with BP, the results are generally comparable for all weight cardinalities. This demonstrates that BP is able to resolve the errors produced by restricting weight cardinality, making it an

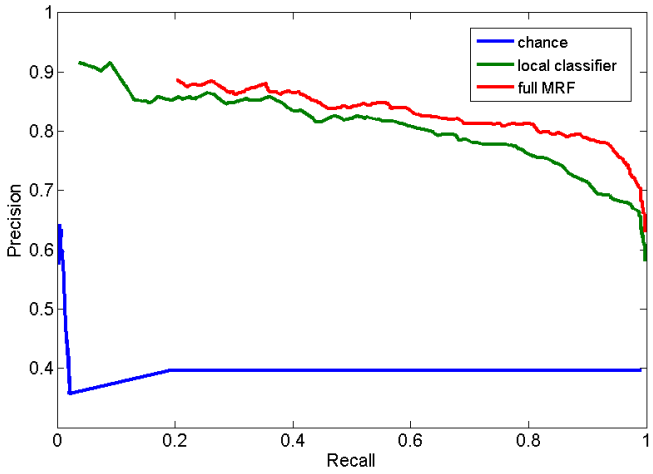


Figure 4.6: **Precision-recall curve:** Precision vs. recall for cell-based error metric, demonstrating improvements over baseline results for navigation purposes.

ideal choice as an inference algorithm in neuromorphic hardware which imposes that constraint.

Spatially-varying models. Baseline results were improved by spatially-varying models in most cases, and full MRF results were improved in all cases except poles. This suggests that spatially varying models are useful in most cases, but might be foregone for obstacles with spatially consistent visual appearance such as poles.

Performance in different environments. Since my algorithm uses supervised learning to learn local classifier weights, it is easily adapted to new obstacle classes. The four classes presented here are very visually different, yet my model is able to detect obstacles effectively in each. My model was also able to handle variations in lighting and obstacle appearance (such as leaves falling off the trees). Figure 4.7 shows an example of a tree trunk classifier using my algorithm performing effectively under a variation in lighting conditions.

Figure 4.7: **Varying lighting:** Classification results for a tree trunk classifier using my algorithm, on the same tree under very different lighting conditions.

Table 4.2: **Robotic experiment results:** Error rates presented for classification, motion execution, and overall successful avoidance

Type	Obstacles	Tests	Success Rate		
			Class.	Motion	Overall
Trees	3	20	100.0%	95.0%	95.0%
Fences	3	25	100.0%	100.0%	100.0%
Poles	2	8	100.0%	87.5%	87.5%
Total	8	53	100.0%	96.2%	96.2%

4.6.2 Robotic Experiments in Real Environments

I performed obstacle avoidance experiments on my aerial robot platform, an AirRobot with a single onboard camera and an onboard IMU for stabilization.[1] Since the hardware described in [92] is still in production, processing was done using an

[1]Because of funding agency's restrictions, I am not allowed to disclose more detailed specifications of my system.

Figure 4.8: **Avoiding obstacles in series:** My aerial robot avoiding a fence and a pole in sequence. Left: Overhead map of area, red indicates obstacles avoided, blue robot path. Robot started at the blue dot, behind and below the level of the fence, moved upwards to a safe altitude, and proceeded over the fence and through the clear area. It then stopped at the pole, detected it as an obstacle, moved diagonally to the right to avoid it, and then forwards again through the following clear area.

offboard laptop computer running a MATLAB simulation of the hardware. Due to transmission latency and a lack of the necessary parallelism, this limited me to classifying a single image at a time, then taking a large step with the robot.

For each experiment, the robot was driven to a hovering position roughly 2-10 meters from the obstacle. Using the robot's IMU, I determined when it was in a stable, level pose, then captured an image from its onboard camera. This image was transmitted to the laptop, which performed the classification described in Section 4.4, using a classifier trained for the appropriate obstacle class. I then used the algorithm described in Section 4.5 to select an appropriate avoidance maneuver. The robot then executed a predetermined, fixed motion plan based on the maneuver chosen.

In Table 4.2, I report three types of success-rates: "classification": when the obstacle classification was correct, "motion": when the computed obstacle avoidance maneuver was correct, and "overall": when the robot successfully executed the maneuver avoiding the obstacle.

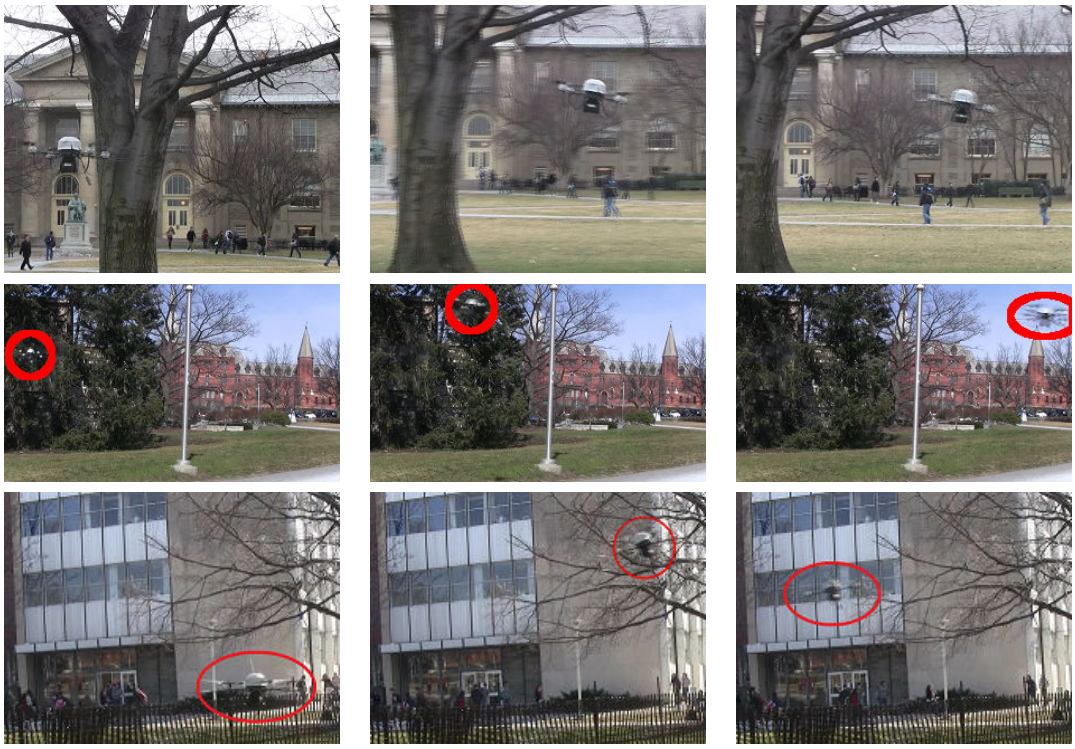


Figure 4.9: **Avoiding obstacles:** AirRobot avoiding obstacles based on my classification results (robot circled in red in cases where it's difficult to see)

In all the experiments in Table 4.2, classification results from my model were of sufficient accuracy to allow the controller described in Section 4.5 to properly select a safe motion. My model identified both foreground obstacles and clear areas consistently in these experiments.

As seen in Figure 4.9, my algorithm worked even in environments with visually cluttered backgrounds such as buildings and background trees. While these backgrounds did cause some false positives, as seen in some cases in Figure 4.5, these were only in the top region of the image, which was not considered by the controller. Since my algorithm works at visual range, it was able to perceive obstacles from long distances, allowing the robot to make large, predetermined motions to avoid them.

Some obstacles encountered were semi-transparent, such as the fence in the bottom row of Figure 4.9. Even though the visual appearance of the fence varied depending on viewing angle and background, and the fence exhibited specular reflections in some places, my algorithm was able to produce an extremely clean labeling of the fence as seen in Figure 4.5.

I also performed an experiment where the robot avoided multiple classes of obstacles in sequence. The classification models for fences and poles were run in parallel, and only motions determined to be safe by both were considered. The robot was re-oriented to its goal travel direction after each motion. The robot was able to travel roughly 25 meters while avoiding obstacles, as shown in Figure 4.8. This approach was able to produce good results because the models were able to correctly report a lack of foreground obstacles when there were none, allowing only the classifiers for which foreground obstacles were present to dictate maneuver selection.

Video showing my robot avoiding obstacles using my algorithm is available at: `http://mav.cs.cornell.edu`

4.7 Conclusions

I presented a learning algorithm that takes as input a single image and outputs an obstacle map. My algorithm uses a Markov Random Field to model the mapping from visual features to obstacles and the relations between neighboring regions in the image. I use parallel neuromorphic hardware for performing inference in the model. This hardware is well-suited for aerial robots because of its extremely low power requirements. My MRF model also considers the cardinality constraints of

the synaptic weights, and tries to minimize the power requirements for inference by minimizing the number of spikes. In upcoming hardware, my algorithms would allow several frames per second to be processed, while consuming less than 1 W of power.

I evaluated my algorithms in both learning experiments and robotic experiments. In learning experiments, my MRF model made significant improvements in classifier accuracy both quantitatively and qualitatively for the purpose of obstacle avoidance. In robotic experiments, my algorithm was able to correctly identify the locations of forgeround obstacles in all tests, allowing the robot to select an evasive maneuver to avoid them. Some of these tests involved multiple obstacles of different classes in sequence, demonstrating that inference results from multiple models can be effectively combined.

4.8 Acknowledgements

I would like to thank Dharmendra Modha, Shyamal Chandra, Thomas Zimmerman, Stefano Carpin, Steve Esser, Myron Flickner, Jerry Yeh, and Dale Cassidy for useful disussions, and Jasdeep Hundal and Brian Wojcik for their help with experiments. This work was partially supported by DARPA under grant #HR0011-09-C-0002 and by an Alfred P. Sloan research fellowship.

CHAPTER 5

CONCLUSION

Here, I presented several works which apply deep learning algorithms and neural networks to robotics problems. These works showed that by identifying hard-to-model nonlinearities and designing deep networks and learning algorithms to model them, we can enable robotic systems to learn to solve a wide range of problems from data. I showed that these networks can model even abstract nonlinearities such as mapping RGB-D image data to graspability in Chapter 2, and complex time-varying nonlinearities like food-cutting dynamics in Chapter 3. These algorithms were able to improve prediction accuracy in both cases, giving more accurate results than even learning approaches with hand-engineered features.

In my work on aerial robot obstacle avoidance in Chapter 4, I showed the promise of up-and-coming neural hardware, which lends itself well to implementing deep learning algorithms. Such hardware would allow these algorithms to operate completely in parallel, allowing us to run hundreds of thousands of neurons at a time while maintaining an update rate fast enough to be useful for real-time robotic applications.

In all these works, I applied these algorithms to real-world robotic systems. I developed three different end-to-end systems which were able to autonomously solve their given tasks. While these systems still required significant engineering efforts to develop, using learning algorithms allowed me to avoid having to model such complex nonlinearities by hand. This shows that these deep learning algorithms are not only capable of producing impressive offline results, but are capable of producing strong results in online, real-world robotic systems.

BIBLIOGRAPHY

[1] P. Abbeel, A. Coates, M. Quigley, and A. Y. Ng. An application of reinforcement learning to aerobatic helicopter flight. In *NIPS*, 2007.

[2] C. G. Atkeson, C. H. An, and J. M. Hollerbach. Estimation of inertial parameters of manipulator loads and links. *Int. J. Rob. Res.*, 5(3):101–119, Sept. 1986.

[3] A. Bachrach, S. Prentice, R. He, and N. Roy. Range - robust autonomous navigation in gps-denied environments. *Journal of Field Robotics*, 28(5): 644–666, 2011.

[4] C. Bartolozzi, F. Rea, C. Clercq, M. Hofstätter, D. Fasnacht, G. Indiveri, and G. Metta. Embedded neuromorphic vision for humanoid robots. In *ECVW*, 2011.

[5] D. Batra and A. Saxena. Learning the right model: Efficient max-margin learning in laplacian crfs. In *CVPR*, 2012.

[6] M. Beetz, U. Klank, I. Kresse, A. Maldonado, L. Mösenlechner, D. Pangercic, T. Rühr, and M. Tenorth. Robotic Roommates Making Pancakes. In *Humanoids*, 2011.

[7] Y. Bengio. Learning deep architectures for AI. *FTML*, 2(1):1–127, 2009.

[8] Y. Bengio, P. Simard, and P. Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 5 (2):157–166, 1994.

[9] S. Bennett. A brief history of automatic control. *Control Systems, IEEE*, 16 (3):17–25, Jun 1996. ISSN 1066-033X.

[10] D. Bhat and S. Nayar. Stereo in the presence of specular reflection. In *ICCV*, 1995.

[11] T. Bhattacharjee, J. Wade, and C. C. Kemp. Material recognition from heat transfer given varying initial conditions and short-duration contact. In *RSS*, 2015.

[12] A. Bicchi and V. Kumar. Robotic grasping and contact: a review. In *ICRA*, 2000.

[13] C. Bills, J. Chen, and A. Saxena. Autonomous mav flight in indoor environments using single image perspective cues. In *ICRA*, 2011.

[14] L. Bo, X. Ren, and D. Fox. Unsupervised Feature Learning for RGB-D Based Object Recognition. In *ISER*, 2012.

[15] J. Bohg, A. Morales, T. Asfour, and D. Kragic. Data-driven grasp synthesis - a survey. accepted.

[16] M. Bollini, J. Barry, and D. Rus. Bakebot: Baking cookies with the pr2. In *IROS PR2 Workshop*, 2011.

[17] D. Bowers and R. Lumia. Manipulation of unmodeled objects using intelligent grasping schemes. *IEEE Trans Fuzzy Sys*, 11(3), 2003.

[18] E. Brunskill, L. Kaelbling, T. Lozano-Pérez, and N. Roy. Continuous-state POMDPs with hybrid dynamics. In *ISAIM*, 2008.

[19] M. V. Butz, O. Herbort, and J. Hoffmann. Exploiting redundancy for flexible behavior: Unsupervised learning in a modular sensorimotor control architecture. *Psychological Review*, 114:1015–1046, 2007.

[20] C. Cadena and J. Kosecka. Semantic parsing for priming object detection in rgb-d scenes. In *ICRA Workshop on Semantic Perception, Mapping and Exploration*, 2013.

[21] S. Chernova and M. Veloso. Confidence-based policy learning from demonstration using gaussian mixture models. In *AAMAS*, 2007.

[22] C.-M. Chow, A. G. Kuznetsov, and D. W. Clarke. Successive one-step-ahead predictions in multiple model predictive control. *Int. J. Systems Science*, 29 (9):971–979, 1998.

[23] A. Coates and A. Y. Ng. Selecting receptive fields in deep networks. In *NIPS*, 2011.

[24] A. Coates, B. Carpenter, C. Case, S. Satheesh, B. Suresh, T. Wang, D. J. Wu, and A. Y. Ng. Text detection and character recognition in scene images with unsupervised feature learning. In *ICDAR*, 2011.

[25] A. Collet Romea, D. Berenson, S. Srinivasa, and D. Ferguson . Object recognition and full pose registration from a single image for robotic manipulation. In *ICRA*, 2009.

[26] A. Collet Romea, M. Martinez Torres, and S. Srinivasa. The moped framework: Object recognition and pose estimation for manipulation. *IJRR*, 30 (10):1284 – 1306, 2011.

[27] R. Collobert and J. Weston. A unified architecture for natural language processing: deep neural networks with multitask learning. In *ICML*, 2008.

[28] R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu, and P. Kuksa. Natural language processing (almost) from scratch. *JMLR*, 12: 2493–2537, 2011.

[29] J. Courbon, Y. Mezouar, N. Guenard, and P. Martinet. Visual navigation of a quadrotor aerial vehicle. In *IROS*, 2009.

[30] M. P. Deisenroth and C. E. Rasmussen. Pilco: A model-based and data-efficient approach to policy search. In *ICML*, 2011.

[31] R. Detry, E. Baseski, M. Popovic, Y. Touati, N. Kruger, O. Kroemer, J. Peters, and J. Piater. Learning object-specific grasp affordance densities. In *ICDL*, 2009.

[32] M. Dogar, K. Hsiao, M. Ciocarlie, and S. Srinivasa. Physics-based grasp planning through clutter. In *RSS*, 2012.

[33] M. Dominici and R. Cortesao. Model predictive control architectures with force feedback for robotic-assisted beating heart surgery. In *ICRA*, 2014.

[34] J. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. *JMLR*, 12:2121–2159, July 2011.

[35] Eigen Matrix Library. `http://eigen.tuxfamily.org`.

[36] S. Ekvall and D. Kragic. Learning and evaluation of the approach vector for automatic grasp generation and planning. In *ICRA*, 2007.

[37] F. Endres, J. Hess, J. Sturm, D. Cremers, and W. Burgard. 3d mapping with an RGB-D camera. *International Journal of Robotics Research (IJRR)*, 2013.

[38] T. Erez, K. Lowrey, Y. Tassa, V. Kumar, S. Kolev, and E. Todorov. An integrated system for real-time model-predictive control of humanoid robots. In *ICHR*, 2013.

[39] C. Ferrari and J. Canny. Planning optimal grasps. *ICRA*, 1992.

[40] A. Foka and P. Trahanias. Real-time hierarchical pomdps for autonomous robot navigation. *Robot. Auton. Syst.*, 55(7):561–571, Jul 2007.

[41] C. R. Gallegos, J. Porta, and L. Ros. Global optimization of robotic grasps. In *RSS*, 2011.

[42] M. Gemici and A. Saxena. Learning haptic representation for manipulating deformable food objects. In *IROS*, 2014.

[43] S. Ghosh and J. Mulligan. A segmentation guided label propagation scheme for autonomous navigation. In *ICRA*, 2010.

[44] R. Girshick, J. Donahue, T. Darrell, and J. Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In *CVPR*, 2014.

[45] C. Goldfeder, M. Ciocarlie, H. Dang, and P. K. Allen. The Columbia grasp database. In *ICRA*, 2009.

[46] I. Goodfellow, Q. Le, A. Saxe, H. Lee, and A. Y. Ng. Measuring invariances in deep networks. In *NIPS*, 2009.

[47] A. Graves, A. Mohamed, and G. E. Hinton. Speech recognition with deep recurrent neural networks. In *ICASSP*, 2013.

[48] H. K. Heidarsson and G. S. Sukhatme. Obstacle detection from overhead imagery using self-supervised learning for autonomous surface vehicles. In *IROS*, 2011.

[49] S. Heshmati-alamdari, G. K. Karavas, A. Eqtami, M. Drossakis, and K. Kyriakopoulos. Robustness analysis of model predictive control for constrained image-based visual servoing. In *ICRA*, 2014.

[50] G. Hinton and R. Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–507, 2006.

[51] T. Howard, C. Green, and A. Kelly. Receding horizon model-predictive control for mobile robot navigation of intricate paths. In *International Conference on Field and Service Robotics*, July 2009.

[52] S. Hrabar. 3d path planning and stereo-based obstacle avoidance for rotorcraft uavs. In *IROS*, 2008.

[53] K. Hsiao, L. P. Kaelbling, and T. Lozano-Pérez. Grasping POMDPs. In *IEEE International Conference on Robotics and Automation*, 2007. URL `http://lis.csail.mit.edu/pubs/tlp/pomdp-grasp-final.pdf`.

[54] K. Huebner and D. Kragic. Selection of Robot Pre-Grasps using Box-Based Shape Approximation. In *IROS*, 2008.

[55] A. Hyvärinen, P. O. Hoyer, and M. Inki. Topographic independent component analysis. *Neural computation*, 13(7):1527–1558, 2001.

[56] A. Hyvärinen, J. Karhunen, and E. Oja. *Principal Component Analysis and Whitening*, chapter 6, pages 125–144. John Wiley & Sons, Inc., 2002.

[57] S. B. i Badia, P. Pyk, and P. F. M. J. Verschure. A fly-locust based neuronal control system applied to an unmanned aerial vehicle: the invertebrate neuronal principles for course stabilization, altitude control and collision avoidance. *IJRR*, 26(7):759–772, 2007.

[58] A. Jain, A. Singh, H. S. Koppula, S. Soh, and A. Saxena. Recurrent neural networks for driver activity anticipation via sensory-fusion architecture. In *Cornell Tech Report*, 2015.

[59] A. Jalali, P. Ravikumar, S. Sanghavi, and C. Ruan. A dirty model for multi-task learning. In *NIPS*, 2010.

[60] N. R. Jared Glover, Daniela Rus. Probabilistic models of object geometry for grasp planning. In *RSS*, 2008.

[61] Y. Jiang, S. Moseson, and A. Saxena. Efficient grasping from RGBD images: Learning using a new rectangle representation. In *ICRA*, 2011.

[62] Y. Jiang, J. R. Amend, H. Lipson, and A. Saxena. Learning hardware agnostic grasps for a universal jamming gripper. In *ICRA*, 2012.

[63] Y. Jiang, M. Lim, C. Zheng, and A. Saxena. Learning to place new objects in a scene. *IJRR*, 31(9), 2012.

[64] T. Jochem, D. Pomerleau, and C. Thorpe . Vision-based neural network road and intersection detection and traversal. In *IROS*, 1995.

[65] I. Kamon, T. Flash, and S. Edelman. Learning to grasp using visual information. In *ICRA*, 1996.

[66] S. Khansari-Zadeh and A. Billard. Learning stable nonlinear dynamical systems with gaussian mixture models. *Robotics, IEEE Transactions on*, 27(5): 943–957, Oct 2011.

[67] D. Kim, J. Sun, S. Min, O. James, M. Rehg, and A. F. Bobick. Traversability classification using unsupervised on-line visual learning for outdoor robot navigation. In *ICRA*, 2006.

[68] J. Kocijan, R. Murray-Smith, C. E. Rasmussen, and A. Girard. Gaussian process model based predictive control. In *American Control Conference*, 2004.

[69] A. Konno, R. Uchikura, T. Ishihara, T. Tsujita, T. Sugimura, J. Deguchi, M. Koyanagi, and M. Uchiyama. Development of a high speed vision system for mobile robots. In *IROS*, 2006.

[70] M. Koval, N. Pollard , and S. Srinivasa. Pre- and post-contact policy decomposition for planar contact manipulation under uncertainty. *IJRR*, August 2015.

[71] D. Kragic and H. I. Christensen. Robust visual servoing. *IJRR*, 2003.

[72] S. Kumar and M. Hebert. Discriminative random fields: A discriminative framework for contextual interaction in classification. In *ICCV*, 2003.

[73] K. Lai, L. Bo, X. Ren, and D. Fox. A large-scale hierarchical multi-view rgb-d object dataset. In *ICRA*, 2011.

[74] K. Lakshminarayana. Mechanics of form closure. *ASME*, 1978.

[75] Q. Le, M. Ranzato, R. Monga, M. Devin, K. Chen, G. Corrado, J. Dean, and A. Ng. Building high-level features using large scale unsupervised learning. In *ICML*, 2012.

[76] Q. V. Le, D. Kamm, A. F. Kara, and A. Y. Ng. Learning to grasp objects with multiple contact points. In *ICRA*, 2010.

[77] Q. V. Le, A. Coates, B. Prochnow, and A. Y. Ng. On optimization methods for deep learning. In *ICML*, 2011.

[78] Y. LeCun, F. Huang, and L. Bottou. Learning methods for generic object recognition with invariance to pose and lighting. In *CVPR*, 2004.

[79] H. Lee, R. Grosse, R. Ranganath, and A. Y. Ng. Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations. In *ICML*, 2009.

[80] H. Lee, Y. Largman, P. Pham, and A. Y. Ng. Unsupervised feature learning for audio classification using convolutional deep belief networks. In *NIPS*, 2009.

[81] S. Levine and P. Abbeel. Learning neural network policies with guided policy search under unknown dynamics. In *NIPS*, 2015.

[82] S. Levine, N. Wagener, and P. Abbeel. Learning contact-rich manipulation skills with guided policy search. *ICRA*, 2015.

[83] C. Li, A. Saxena, and T. Chen. θ-mrf: Capturing spatial and semantic structure in the parameters for scene understanding. In *NIPS*, 2011.

[84] J. Luo and K. Hauser. Robust trajectory optimization under frictional contact with iterative learning. In *RSS*, 2015.

[85] J. Maitin-shepard, M. Cusumano-towner, J. Lei, and P. Abbeel. Cloth grasp point detection based on multiple-view geometric cues with application to robotic towel folding. In *ICRA*, 2010.

[86] J. Maitin-Shepard, J. L. M. Cusumano-Towner, and P. Abbeel. Cloth grasp point detection based on multiple-view geometric cues with application to robotic towel folding. In *ICRA*, 2010.

[87] J. Malik, S. Belongie, J. Shi, and T. Leung. Textons, contours and regions: Cue integration in image segmentation. In *ICCV*, 1999.

[88] C. McFarland and L. Whitcomb. Experimental evaluation of adaptive model-based control for underwater vehicles in the presence of unmodeled actuator dynamics. In *ICRA*, 2014.

[89] T. McGee, R. Sengupta, and K. Hedrick. Obstacle detection for small autonomous aircraft using sky segmentation. In *ICRA*, 2005.

[90] D. Mellinger, N. Michael, and V. Kumar. Trajectory generation and control for precise aggressive maneuvers with quadrotors. In *ISER*, Dec 2010.

[91] R. Memisevic and G. E. Hinton. Learning to represent spatial transformations with factored higher-order boltzmann machines. *Neural Computation*, 22(6):1473–1492, June 2010.

[92] P. Merolla, J. Arthur, F. Akopyan, N. Imam, R. Manohar, and D. Modha. A digital neurosynaptic core using embedded crossbar memory with 45pj per spike in 45nm. In *CICC*, 2011.

[93] T. Merz and F. Kendoul. Beyond visual range obstacle avoidance and infrastructure inspection by an autonomous helicopter. In *IROS*, 2011.

[94] J. Michels, A. Saxena, and A. Y. Ng. High speed obstacle avoidance using monocular vision and reinforcement learning. In *ICML*, 2005.

[95] A.-R. Mohamed, G. Dahl, and G. E. Hinton. Acoustic modeling using deep belief networks. *IEEE Trans Audio, Speech, and Language Processing*, 20 (1):14–22, 2012.

[96] F. C. Moon and T. Kalmár-Nagy. Nonlinear models for complex dynamics in cutting materials. *Philosophical Transactions of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences*, 359(1781):695–711, 2001.

[97] R. Moore, S. Thurrowgood, D. Bland, D. Soccol, and M. Srinivasan. A stereo vision system for uav guidance. In *IROS*, 2009.

[98] A. Morales, P. J. Sanz, and Àngel P. del Pobil. Vision-based computation of three-finger grasps on unknown planar objects. In *IROS*, 2002.

[99] R. Mudra, R. Hahnloser, and R. J. Douglas. Neuromorphic active vision used in simple navigation behavior for a robot. In *Proc. 7th Int. Conf. On Microelectronics for Neural Networks*, pages 7–9, 1999.

[100] K. P. Murphy, Y. Weiss, and M. I. Jordan. Loopy belief propagation for approximate inference: An empirical study. In *UAI*, 1999.

[101] I. Na, S. H. Han, and H. Jeong. Stereo-based road obstacle detection and tracking. In *ICACT*, 2011.

[102] B. Nabbe and M. Hebert. Extending the path-planning horizon. volume 26, pages 997–1024, 2007.

[103] K. S. Narendra and A. M. Annaswamy. *Stable Adaptive Systems*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989. ISBN 0-13-839994-8.

[104] J. Ngiam, A. Khosla, M. Kim, J. Nam, H. Lee, and A. Y. Ng. Multimodal deep learning. In *ICML*, 2011.

[105] V. Nguyen. Constructing stable force-closure grasps. In *ACM Fall joint computer conf*, 1986.

[106] D. Nguyen-Tuong and J. Peters. Model learning for robot control: a survey. *Cognitive Processing*, 12(4), 2011.

[107] M. Osadchy, Y. LeCun, and M. Miller. Synergistic face detection and pose estimation with energy-based models. *JMLR*, 8:1197–1215, 2007.

[108] C. Papazov, S. Haddadin, S. Parusel, K. Krieger, and D. Burschka. Rigid 3d geometry matching for grasping of known objects in cluttered scenes. *IJRR*, 31(4):538–553, Apr. 2012.

[109] J. Pearl. Fusion, propagation, and structuring in belief networks. *Artif. Intell.*, 29(3):241–288, 1986.

[110] J. H. Piater. Learning visual features to predict hand orientations. In *ICML*, 2002.

[111] C. Plagemann, F. Endres, J. M. Hess, C. Stachniss, and W. Burgard. Monocular range sensing: A non-parametric learning approach. In *ICRA*, 2008.

[112] F. T. Pokorny, K. Hang, and D. Kragic. Grasp moduli spaces. In *RSS*, 2013.

[113] J. Ponce, D. Stam, and B. Faverjon. On computing two-finger force-closure grasps of curved 2D objects. *IJRR*, 12(3):263, 1993.

[114] M. L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley-Interscience, 1994.

[115] C. E. Rasmussen and H. Nickisch. GPML Matlab Code version 3.5. `http://www.gaussianprocess.org/gpml/code/matlab/doc/`.

[116] M. Riedmiller, T. Gabel, R. Hafner, and S. Lange. Reinforcement learning for robot soccer. *Autonomous Robots*, 27(1):55–73, 2009.

[117] A. Rodriguez, M. Mason, and S. Ferry. From caging to grasping. In *RSS*, 2011.

[118] ROS: Robot Operating System. `http://www.ros.org`.

[119] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation. In *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, pages 318–362. MIT Press, 1986.

[120] R. B. Rusu, N. Blodow, and M. Beetz. Fast point feature histograms (FPFH) for 3D registration. In *ICRA*, 2009.

[121] R. B. Rusu, G. Bradski, R. Thibaux, and J. Hsu. Fast 3D recognition and pose using the viewpoint feature histogram. In *IROS*, 2010.

[122] A. Sahbani, S. El-Khoury, and P. Bidaud. An overview of 3d object grasp synthesis algorithms. *Robot. Auton. Syst.*, 60(3):326–336, Mar. 2012.

[123] A. Saxena, S. Chung, and A. Ng. Learning depth from single monocular images. In *NIPS*, 2005.

[124] A. Saxena, J. Driemeyer, J. Kearns, and A. Ng. Robotic grasping of novel objects. In *NIPS*, 2006.

[125] A. Saxena, S. Chung, and A. Ng. 3-d depth reconstruction from a single still image. *IJCV*, 76(1):53–69, 2008.

[126] A. Saxena, J. Driemeyer, and A. Y. Ng. Robotic grasping of novel objects using vision. *IJRR*, 27(2):157–173, 2008.

[127] A. Saxena, M. Sun, and A. Y. Ng. Make3d: Depth perception from a single still image. In *AAAI*, 2008.

[128] A. Saxena, L. L. S. Wong, and A. Y. Ng. Learning grasp strategies with partial shape information. In *AAAI*, 2008.

[129] A. Saxena, M. Sun, and A. Ng. Make3d: Learning 3d scene structure from a single still image. *PAMI*, 2009.

[130] A. Saxena, A. Jain, O. Sener, A. Jami, D. K. Misra, and H. S. Koppula. Robo brain: Large-scale knowledge engine for robots. *Tech Report*, Aug 2014.

[131] P. Sermanet, K. Kavukcuoglu, S. Chintala, and Y. LeCun. Pedestrian detection with unsupervised multi-stage feature learning. In *CVPR*. 2013.

[132] D. Shim, H. Kim, and S. Sastry. Decentralized reflective model predictive control of multiple flying robots in dynamic environment. In *Conference on Decision and Control*, 2003.

[133] K. B. Shimoga. Robot grasp synthesis algorithms: A survey. *IJRR*, 15(3): 230–266, June 1996.

[134] R. Socher, B. Huval, B. Bhat, C. D. Manning, and A. Y. Ng. Convolutional-recursive deep learning for 3D object classification. In *NIPS*, 2012.

[135] B. Sofman, E. L. Ratliff, J. A. D. Bagnell, J. Cole, N. Vandapel, and A. T. Stentz. Improving robot navigation through self-supervised online learning. *Journal of Field Robotics*, 23(12), 2006.

[136] K. Sohn, D. Y. Jung, H. Lee, and A. Hero III. Efficient learning of sparse, distributed, convolutional feature representations for object recognition. In *ICCV*, 2011.

[137] S. P. Soundararaj, A. K. Sujeeth, and A. Saxena. Autonomous indoor helicopter flight using a single onboard camera. In *IROS*, 2009.

[138] N. Srivastava and R. Salakhutdinov. Multimodal learning with deep Boltzmann machines. In *NIPS*, 2012.

[139] J. Sung, S. H. Jin, and A. Saxena. Robobarista: Object part-based transfer of manipulation trajectories from crowd-sourcing in 3d pointclouds. In *Cornell Tech Report*, 2015.

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

[141] I. Sutskever, J. Martens, and G. Hinton. Generating text with recurrent neural networks. In *ICML*, 2011.

[142] C. Szegedy, A. Toshev, and D. Erhan. Deep neural networks for object detection. In *NIPS*. 2013.

[143] G. W. Taylor and G. E. Hinton. Factored conditional restricted boltzmann machines for modeling motion style. In *ICML*, 2009.

[144] C. Teuliere and E. Marchand. Direct 3d servoing using dense depth maps. In *IROS*, 2012.

[145] V. Vapnik. *Statistical Learning Theory*. John Wiley and Sons, 1998.

[146] P. Viola and M. Jones. Rapid object detection using a boosted cascade of simple features. In *CVPR*, 2001.

[147] Z. Wang, A. Boularias, K. Muelling, B. Schoelkopf, and J. Peters. Anticipatory action selection for human-robot table tennis. accepted.

[148] J. Weisz and P. K. Allen. Pose error robust grasping from contact wrench space metrics. In *ICRA*, 2012.

[149] T. Whelan, H. Johannsson, M. Kaess, J. Leonard, and J. McDonald. Robust real-time visual odometry for dense RGB-D mapping. In *ICRA*, 2013.

[150] K. M. Wurm, R. Kümmerle, C. Stachniss, and W. Burgard. Improving robot navigation in structured outdoor environments by identifying vegetation from laser data. In *IROS*, 2009.

[151] L. Zhang, M. Ciocarlie, and K. Hsiao. Grasp evaluation with graspable feature matching. In *RSS Workshop on Mobile Manipulation*, 2011.

[152] S. Zingg, D. Scaramuzza, S. Weiss, and R. Siegwart. Mav navigation through indoor corridors using optical flow. In *ICRA*, 2010.