Freie Universität Berlin

Learning Representations from Motion Trajectories: Analysis and Applications to Robot Planning and Control

Dissertation zur Erlangung des Grades
eines Doktors der Naturwissenschaften (Dr. rer. nat.)
am Fachbereich Mathematik und Informatik
der Freien Universität Berlin

von

Nikolay Nikolaev Jetchev

Berlin
January 31, 2012

FREIE UNIVERSITÄT BERLIN
Fachbereich Mathematik und Informatik

Institut für Informatik
Machine Learning and Robotics Lab
Prof. Dr. Marc Toussaint

Learning Representations from Motion Trajectories: Analysis and Applications to Robot Planning and Control

Dissertation

Autor Nikolay Nikolaev Jetchev
aus Sofia

Vorgelegt im 2. Februar 2012

Disputation am 20. April 2012

Erstgutachter Prof. Dr. Marc Toussaint
Zweitgutachter Prof. Dr. Klaus-Robert Müller

Summary

Generating motion is a crucial aspect of articulated robotics. Many robot manipulation tasks can be defined and solved as a motion trajectory generation problem where the robot needs to calculate and execute a task-appropriate movement. Multiple efficient methods have been developed for this problem in the robotic community, but they do not make use of the patterns found in motion data. We will propose in this thesis novel approaches to combine machine learning and robotics.

The main conceptual achievement of this thesis is the successful fusion of machine learning and robotic algorithms to improve motion generation and discover the structure of different motion tasks. By observing examples of robot motions, optimal for a given task, we can find the relevant features of the motions, and discover the latent structure inherent in the interaction between robot and workspace. This leads to new algorithms for planning and control which are demonstrated in numerous experiments to improve on previous approaches in terms of speed and generalization ability.

For speeding up motion planning we developed an algorithm called *Trajectory Prediction*. In the trajectory planning scenario the desired robot behavior is specified by a cost function and a planner algorithm is used to generate low-cost motions. Our contribution is to predict an appropriate initial trajectory that can speed up the planner. We do this by learning a mapping from situation to trajectory, and extracting the representations useful for such a mapping.

For learning from demonstration we developed an algorithm called *Task Space Retrieval Using Inverse Optimal Control*. In this scenario no cost function is available to specify what is a good motion. By observing example trajectories our method can learn a value function model and an efficient sparse task space representation of the desired behavior. A controller for motion generation is developed based on this value function, effectively generalizing the demonstrated behavior in novel situations.

Acknowledgements
Danksagung

I have been very lucky to have Marc Toussaint as my main thesis advisor. Throughout my time as PhD student he has always supported me and inspired me with his creative vision and great expertise in various fields. His visionary ideas and helpful approach allowed me to realize the joy of discovery and improve myself as a thinker. This gave me the chance to vastly broaden my horizons and meet and discuss science and life with fascinating people.

I thank Klaus-Robert Müller for being the second referee of my thesis. His insights in machine learning have helped me the appreciate the discipline on a deeper level. I am very happy that I had the chance to be a part of the great IDA lab and enjoy the amazingly open and creative atmosphere there.

I owe a lot to my mother Ekaterina for her wise advice.

I thank Denitza for staying patient with me and always smiling.

And I want to mention the TU and FU colleagues I had the pleasure to know: Tobias, Stanio, Daniel, Martijn, Claudia, Katja, Pascal, Gregoire and all the rest. We had good times together.

Berlin, my favorite city.

"Our nature consists in motion; complete rest is death."

Blaise Pascal

"The aim of every artist is to arrest motion, which is life, by artificial means and hold it fixed so that a hundred years later, when a stranger looks at it, it moves again since it is life."

William Faulkner

"Data is a precious thing and will last longer than the systems themselves."

Tim Berners-Lee

"Data is not information, information is not knowledge, knowledge is not understanding, understanding is not wisdom."

Clifford Stoll

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Robots have fascinated people in the last few decades. People have dreamed of having machine companions with which to play games and solve problems in the world together. However, before such high-level concepts become reality, the more basic but at least as challenging task of teaching robots mastery of their bodies and environments should be addressed. So far robots (more exactly, robot arms) perform well in highly structured environments like factories where every object target for manipulation is always in the same predefined position, and a fixed sequence of motion commands is to be executed. However, if somebody changes the positions of the targets the robots cannot adapt as robustly as humans do. In order for robots to become really efficient help in unstructured environments, there need to be improvements in the way the robots comprehend their environment and manipulate the objects in it. The fields of robot motion planning and control deals with generating motion commands that can be executed by robots. This thesis will propose methods that improve on the standard motion algorithms using ideas from the field of Machine learning.

Many of the planning and control algorithms have strong priors about the robot motions to be generated, which are chosen by a human designer beforehand. If one is creating a motion for a new task for the first time from "scratch" this is an appropriate approach. However, one can learn much better priors from data. We will replace some of the default choices used for motion generation (default straight initialization or controllers in a fixed task space) with data driven modules that can lead to better performance and discover the structure of the tasks that often needs to be guessed by a human designer. We reason that the ability to reflect upon the motion experience of oneself or the others is an invaluable tool to draw conclusions about the future. This is valid for motor imagery, representations and perception of the world.

> Motor cognition is mental processing in which the motor system draws on stored information to plan and produce our own actions, as well as to anticipate, predict, and interpret the actions of others. *Smith and Kosslyn 2007*

We take this view of motor cognition as inspiration for our work and will show throughout this thesis concrete ways in which to use such reflection upon gathered motion tra-

jectory data to reason about the future and generate motions utilizing this experience.

We assume to have access to example trajectories, usually the output from some other motion generation algorithm or recorded from a human demonstration. We also assume that we can represent the motions and situations using some motion features. Usually rich information is available from the robot kinematics and (optionally) additional external sensors for localizing objects in the workspace. Using our methods, we will find better representations expressing the essence of tasks at hand, and improve on the basic (without data and machine learning) versions of popular motion generation methods. The information box below summarizes our main contributions.

Our **contributions** are:

- formulations of robot motion generation problems as novel **machine learning problems** making use of robot experience
  "gather data, define models and training criteria to learn from this information"
- sparse **representations** of motions and situations are learned
  "the most efficient motion features are found from experience"
- the **structure** of motions and situations is extracted, analyzed and visualized using novel techniques for understanding motion trajectories
  "data analysis to understand the interactions between robot and workspace"
- novel **planning and control algorithms**
  "use the learned representations in robot applications"

1.1 Speeding-up Planning

Motion planning is a fundamental issue in articulated robotics. Chapter 3 of this thesis will describe a method that can speed up motion planning by improving the initialization used in stochastic optimal control planners. This is a sensitive aspect of such local planners: they can fall in multiple local optima and a good solution is not guaranteed. Using the structure of encountered environments can provide hints about movements that are likely to be good in a given world configuration.

The animal and human ability to generate trajectories quickly is amazing. In typical every-day situations humans do not seem to require time for motion planning but execute complex trajectories instantly. This suggests that there exists a "reactive trajectory policy" which maps "the situation" (or at least motion relevant features of the situation) to the whole trajectory.[1]

Consider the following example in Figure 1.1: a scenario with a few obstacles and a target (indicated by a red ball). A naive motion prior will initialize the planner with

[1] This is not to be confused with a reactive controller which maps the current sensor state to the current control signal – such a (temporally local) reactive controller could not explain trajectories which efficiently circumvent obstacles in an anticipatory way, as humans naturally do in complex situations.

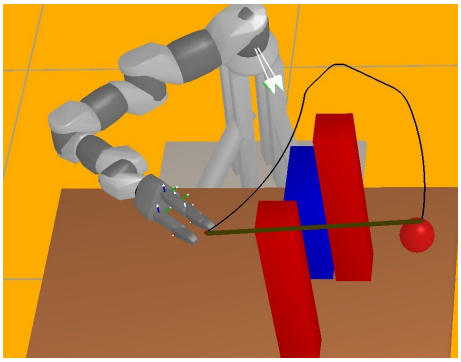


Figure 1.1: A task for motion planning: calculate a trajectory to the target avoiding the obstacles. An intelligent initial plan can speed the whole process significantly.

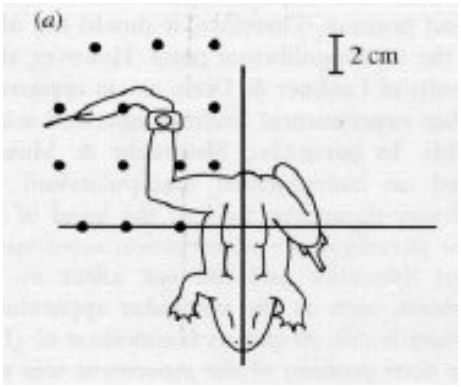


Figure 1.2: Biologically inspired motion primitives: stimulating a frog leads to a hind leg movement with specific attractor, which is a type of motion trajectory. Image taken from (Mussa-Ivaldi and Bizzi, 2000)

a straight-line trajectory going to the target, which can be bad for convergence because of the obstacle collisions. An agent with experience of reaching around obstacles and with an appropriate situation representation can reason about the workspace and which areas are blocked and which are free, and quickly generate as a first motion hypothesis a trajectory avoiding the obstacles in the free region of space. This has some interesting analogies with research in the area of motor neural control (Mussa-Ivaldi and Bizzi 2000) and human hand motion analysis (Sanger 2000). There is evidence that animals have developed motor primitives, sequences of motion that are executed as whole chunks. If a frog is stimulated electrically, its leg will follow a motion trajectory to a target in the workspace, see Figure 1.2. Different electrical stimulations lead to motions to different targets, and each of these trajectories can be viewed as motor primitives present in the frog and activated by stimulation. These primitives can be combined with each other, resulting in more complex behaviors. Using such motor primitives is popular in imitation learning (Schaal et al. 2003; Ilg et al. 2004) as a representation that allows to model and synthesize motion from examples.

Our approach would also use stored trajectories (similar to primitives) and will learn when to call the appropriate primitive in order to improve planning costs. Such a mapping (if optimal) is utterly complex: the output is not a single current control signal but a whole trajectory which, traditionally, would be the outcome of a computationally expensive trajectory optimization process accounting for collision avoidance, smoothness and other criteria. The input is the current situation, in particular the position of relevant objects, for which it is unclear which representation and coordinate systems to use as a descriptor. We coin this problem *Trajectory Prediction* (TP) and its goal is to learn such an (approximate) mapping from data of previously optimized trajectories in old situations to good trajectories in new situations. We will present TP in Chapter 3 in detail. We proceed with a brief overview of relevant motion planning and learning methods.

1.1.1 Related Motion and Trajectory Generation Methods

Local Planning Methods

Movement generation, one of the most basic robotic tasks, is often viewed as an optimization problem that aims to minimize a cost function. There are many different methods for local trajectory optimization which use the cost gradient information for minimization. Popular approaches use spline-based representation and gradient descent (Zhang and Knoll 1995), covariant gradient descent (Ratliff et al. 2009b), Differential Dynamic Programming (DDP) described by (Dyer and McReynolds, 1970; Atkeson, 1993), a variant of DDP called iterated Linear Quadratic Gaussian (iLQG) from (Todorov and Li, 2005), and Bayesian inference (Toussaint 2009). Such methods are usually fast and can obtain movements of good quality, suitable for control of complex hardware robots with many DoF. However, these local methods can get stuck in local optima. TP aims to predict directly good trajectories such that local planners only need to refine them.

Rapidly-exploring Random Trees (RRT) and Other Sampling Methods

Another approach for finding good movement trajectories is sampling to find obstacle free paths in the configuration and work space of the robot, i.e. finding an appropriate initialization of the movement plan. Popular methods for planning feasible paths without collisions are RRTs (LaValle, 2006) and probabilistic road maps (Kavraki et al., 1995), where random sampling is used to build networks of feasible configuration nodes. These methods are powerful and can find difficult solutions for motion puzzles, but also have the disadvantage to be too slow for high-dimensional manipulation problems. Building an RRT takes some time, and a path to the target in such a network often requires additional optimization to derive an optimal robot trajectory. TP works also in conjunction with a motion planner for refinement of some initial motion. However, TP uses machine learning methods (classification) to predict a motion, which is much faster than the sampling approach of RRTs.

1.1.2 Previous Use of Machine Learning Techniques to Speed up Planning

Transfer in Reinforcement Learning

Concerning our problem of learning from previous optimization data, there exist multiple branches of related work in the literature. In the context of Reinforcement Learning the transfer problem has been addressed, where the value function (Konidaris and Barto 2006) or directly the policy (Peshkin and de Jong 2002) is transferred to a new Markov Decision Process. (Konidaris and Barto, 2006) discussed the importance of representations for the successful transfer. Although the problem setting is similar, these methods are different in that they do not consider a situation descriptor (or features of the "new" MDP) as an input to a mapping which directly predicts the new policy or value function.

Robot Motion Databases

Related work with respect to exploiting databases of previous trajectories has been proposed in the context of RRTs. (Branicky et al., 2008) constructed a compact database of collision free paths that can be reused in future situations to speed up planning under the assumption that some of the previous paths will not be blocked by future obstacles and can be reused for fast planning. (Martin et al., 2007) attempted to bias RRTs such that after planning in a set of initial environments, the obstacles can be rearranged and previous knowledge will be used for faster replanning in the new scene; an environment prior, that visits with higher probability states visited in previous trials, is used to speed up planning and use less tree nodes to achieve the final goal. In both cases, the notion of our situation descriptor and the direct mapping to an appropriate new trajectory is missing.

Another interesting way to exploit a database of previous motions is to learn a "capability map", i.e., a representation of a robot's workspace that can be reached easily, see (Zacharias et al., 2007). While this allows to decide whether a certain task position can be reached quickly, it does not encode a prediction of a trajectory in our sense.

(Stolle and Atkeson, 2007) predict robot locomotion movements for navigation in new situations using databases of state-action pairs to make small steps ahead. In a sense, such use of a database presents action primitives extracted from data similar to TP. However, unlike TP, (Stolle and Atkeson, 2007) adapt their algorithm specifically to the locomotion navigation domain by combining local step planning with global graph-based search, and does not learn data-driven situation feature representations.

The field of imitation learning (Argall et al. 2009) encompasses many approaches using demonstrated motions to learn behaviors: policies that map from situations to actions. The focus is usually to extract motions from human demonstration of different tasks which can be later repeated "exactly" by robots, e.g. see (Calinon and Billard, 2005; Shon et al., 2007) . The demonstrations, often complex gestures or manipulations, are to be repeated accurately, possibly with some robustness to perturbation. However, generalization to different environments and collision avoidance with obstacles there is rarely considered in the imitation process. This is not surprising, since acquiring data in an

interactive way is costly and limits the variation of situations and motions that can be encountered. In Section 3.6 this comparison between TP and imitation learning will be discussed in more detail.

TP approaches motion planning problems in a framework to improve the convergence of local motion planners by predicting situation-appropriate motions. TP predicts whole trajectories at once, not requiring additional global search routines. It seems reasonable that good paths will go around obstacles, and TP can potentially provide a way to start the motion planning task with a path avoiding collisions, similar to RRT. However, our prediction method will not be limited to obstacle avoidance only: TP will predict motion trajectories that improve the convergence of local planners and deal with all aspects implicit in a low cost.

1.2 Imitation Learning

Imitation is a fundamental approach in the animal world. Creatures learn important skills by observing and repeating the behavior of their parents. This has been examined in multiple species, including monkeys, rats and dolphins, see (Krutzen et al., 2005; Voelkl and Huber, 2007). Many animals will copy behavior from *other* species out of curiosity, see Figure 1.3 for a classic example of behavior "mimicry"(Ferrari et al. 2003; Call and Carpenter 2002). There is also extensive neuroscience research about the mirror neurons and their role in imitation behavior (Dinstein et al. 2008). It is assumed that observing motion, predicting its intent on the basis of this observation, and imitating motion are all basic skills required for evolutionary survival (Smith and Kosslyn 2007).

Imitation learning is an important tool for training robots in complex tasks (Argall et al., 2009). Using machine learning techniques to find structure and learn policies from demonstrations of desired behavior is often much more efficient than hand-crafting robot motion controllers or specifying reward functions which to optimize. However, the utility of the behaviors learned in this way is still limited by the implicit assumptions made by the human designers. The question of "what to imitate", i.e. which aspects of the observed motions should be duplicated, is not answered in general.

Consider a robot student trying to learn the forehand swing by observing demonstrations of a teacher (Figure 1.4). There is no obvious way to describe what is a good forehand hit, so we cannot use some planning or optimization to make such a motion. The teacher knows intuitively what is a good hit and can give rewards to the pupil when he performs well or some negative rewards when he fails. However, it is tiring to repeat this reward feedback routine for hundreds of times during a training session. It would be much more practical to just demonstrate the hit a few times and then let the student figure it out. Blindly mirroring the motion (mimicry) is not a good idea, because every game the ball will be approaching the player with different speed and angle. It is also not obvious in what representation should the repeating of the teacher's motion be done. Is the angle of the elbow the important aspect, or the angle of the wrist? Is it the relative position of the racket tip with respect to the ball, or with respect to some other body part of the player? The issue of "generalizing" the observation is far from trivial, and needs

Figure 1.3: The old saying "Monkey see Monkey do" is well illustrated by this monkey mirroring the actions and gestures of a human demonstrator. Image taken from (Ferrari et al., 2003).

Figure 1.4: A task for imitation learning: acquire a skill (tennis forehand) by observing its execution by a proficient teacher. The image (courtesy of `www.qj.net`) shows a character from the *Virtua Tennis 3* computer game.

to be solved if learning by demonstration is to progress beyond simple automated tasks where repetition suffices. A possible solution in this case would be to make a motion so that there is contact with the ball at a certain time and at a location with certain geometrical properties, e.g. displacement relative both to the ball, position of the player's feet, arm, hand, wrists, etc. However, such a ball contact location description is already not trivial: there are a myriad of possible geometrical frames relative to any part of the body we can think of, and it is not directly obvious which to take. It is up to a good student of tennis to "extract" a proper representation, because the teacher will often not provide such an explicit task description.

Extracting task spaces from data is particularly relevant in the context of articulated robotics. In order to generate complex movements for a certain task, a controller typically minimizes costs in a special movement representation or with respect to multiple task features simultaneously (e.g., collision avoidance, hand positioning, finger align-

ment, etc). Which features are suitable and how they are weighted depends on the task at hand. The target of a good motion is typically not a specific configuration state, but a whole task manifold. The challenge in imitation learning thus becomes to retrieve the latent movement representation (which task features are used in the controller) rather than to repeat a point-to-point movement. As an example consider a robot grasping an object: from observing successful motions we should learn that some finger configurations relative to the object surface result in good motions and use such representations in the controller, rather than fix targets in the direct robot configuration space.

Task Space Retrieval Using Inverse Feedback Control (TRIC) addresses the above issues by discovering from data a sparse discriminative value function using the most relevant motion features and using them to generate motions. We mention briefly few of the advantages of TRIC. First, it can handle example demonstrations in high dimensional spaces and select the important features for movement. Second, it can generalize well to situations and constraints unseen during demonstrations (e.g. grasping objects on different positions and collision avoidance with new obstacles), because the learned cost function defines a task manifold. We will present this method in detail in Chapter 5.

1.2.1 Related Work in Imitation Learning

Imitation learning via Direct policy learning (DPL) (Pomerleau, 1991) is a popular approach for learning from demonstration. In essence DPL takes demonstration data in the form of motion trajectories and uses supervised learning to estimate a controller that would reproduce the trajectories. This works well when the exact motion reproduction is desired (gestures, point-to-point motions), but it can have difficulties to generalize and modify the behaviors in new situations. Dynamic Movement Primitives (Schaal et al., 2003) are an advanced method that learns parameters of a dynamic system which defines the controller. However, it still requires to pre-specify a task space in which the attractor operates.

Inverse Optimal Control (IOC) (Ratliff et al., 2006), also known as Inverse Planning or Inverse Reinforcement Learning, takes demonstrations in some state space, and learns a state cost function that gives rise to a policy consistent with this data. IOC provides a general framework to retrieve latent objectives (rewards or costs) in observed behavior. (Ratliff et al., 2009a) describes a combination of IOC and DPL.

1.2.2 Previous Work in Recovering Task Spaces

The question of what are suitable representations of a physical configuration, in particular suitable coordinate systems, has previously been considered in a number of works. (Wagner et al., 2004) discussed the advantages of egocentric versus allocentric coordinate systems for robot control, and (Hiraki et al., 1998) talked about such coordinates in the context of robot and human learning. In the human motion experiments of (Berniker and Kording, 2008) the space of joint angles Q is called intrinsic coordinate system, and a relative position space is called extrinsic.

(Muehlig et al., 2009; Billard et al., 2004) examine different task spaces for robot motions and select the best ones for reproducing different tasks. However, both examine only a small pool of possible task spaces, whereas our method can find rich motion representations from high dimensional task spaces.

Our approach TRIC jointly addresses the problems of finding relevant features of the motion, learning the behavior to imitate, and retrieving a (latent in the demonstrations) value function leading to such behavior. This is a different approach for motion feature selection than looking at data variance in an unsupervised way (Jenkins and Matarić, 2004; Calinon and Billard, 2007). For some applications such an approach can deliver reasonable results, but often it delivers motion features unsuitable for control.

1.3 Outline and Contributions

The thesis follows an outline that gradually introduces the reader to the subject of motion generation, starting from basic techniques and moving up to the advanced novel ML algorithms that make the core of this work.

- **Chapter 2: Robot Motion Background** will present the basics required to model kinematic robot motion generation problems, which will be necessary to understand the subsequent chapters. We will explain the notions of configuration space, workspace and task spaces, which are used to formalize motion generation. We will explain what is meant by robot motion planning, and present briefly two major classes of algorithms, local planners and sampling-based planners, used to create robot motion plans. We will also introduce motion rate control (also known as reactive control or Inverse Kinematics) and the basic equations behind it. We will also define more formally learning from demonstration, and present DPL and IRL, two basic approaches to the problem of repeating and generalizing motion demonstrations.

- **Chapter 3: Trajectory Prediction: Mapping Situations to Motions** will present the first novel algorithm of this thesis: TP, used to map workspace situations to motion trajectories, which can be used to initialize and speed-up local planning methods. We will gradually present the novel concepts involved in the TP algorithm, namely situation representations, Task Space IK Transfer of trajectories between situations, and machine learning methods to predict low-cost motions given situations. We introduce novel techniques and representations for dealing with motion planning. They lead both to faster planning algorithms and better analysis of the structure of motions and situations. ***Contributions:***

  - *We define **the framework of TP**, which is a novel way to use data-driven initializations for local planners*
  - *We propose different **situation descriptors** - sensor and geometry based - appropriate for different information contexts*

  - *We present **IK Task Space Transfer**, a technique for adapting a motion from a database to a new situation, and reason about the properties of task spaces that make them useful for transfer.*
  - *We propose two different **predictive models** for cost-sensitive classification of motions likely to improve local planner performance*
  - *We **discuss** the novel design of the TP framework and why it is better suited than imitation learning to improve local planning*
  - *We give **extensive results** in simulation that show the utility of TP for making motion planning an order of magnitude faster, experiment with a variety of motion planning tasks, and illustrate the important **sparse** aspects of situations for planning motion.*
  - *We describe also a **robot hardware experiment** which showed successfully the application of TP in a real scenario.*

- **Chapter 4: An Extension of Trajectory Prediction: Parallel Process Planning** will present an extension of TP that combines an online version of local planning algorithms with multiple parallel processes each of which uses a different prediction from TP. This proves to be of great practical utility for situations requiring fast reaction to moving objects, for which usual planning algorithms are too slow. ***Contributions**:*

  - *We describe an **online** planning algorithm building upon local planning algorithms that can maintain **multiple local online planners** in parallel and select the best one to control the robot.*
  - *We utilize a **predictive initialization module** using TP to speed-up planning convergence of each of the local planner threads.*

- **Chapter 5: Task Space Retrieval Using Inverse Optimal Control** will introduce a novel algorithm for learning from demonstration, that can use a set of teacher demonstrations as data and learn a controller to imitate them, automatically selecting the features of motion that allow the best generalization of motion. ***Contributions**:*

  - *We define **the novel algorithm of TRIC**, combining ideas from inverse optimal control and discriminative learning, which can be used for efficient **imitation and generalization** of demonstrated motions.*
  - *We define a **novel training loss** to learn a **value function** that is consistent with the demonstrations*
  - *The value function has **generative**, **discriminative** and **sparse** properties.*
  - *We will **discuss** how TRIC compares to other imitation learning methods.*
  - *We give **qualitative and quantitative** results showing the effect of different parameters on the performance of TRIC for motion rate control.*

  - *We introduce **rich geometrical motion features** and use TRIC with **regularization** to **select** the important features.*
  - *We analyze the extracted **sparse** motion features and illustrate their effect on the motion policy, giving an **interpretation of the relevant latent motion aspects**.*
  - *We analyze the **learned invariant representation** of the task goal.*

- **Chapter 6: Conclusions** will briefly summarize all methods and contributions of this thesis, and then sketch an application scenario where our methods can be useful for robot manipulation in a factory. We also consider future research directions combining the approaches of TP and TRIC.

- **Appendix A** contains the proofs of multiple propositions introduced in the thesis.

1.3.1 Publication Summary

This thesis enhances our previously published work with deeper analysis and numerous additional results. The TP method has been first described in (Jetchev and Toussaint, 2009) and then applied to a more realistic hardware and sensor scenario in (Jetchev and Toussaint, 2010). The TRIC method has been originally described in (Jetchev and Toussaint, 2011a). In (Gienger et al., 2008) the question of task manifolds in conjunction with planning is examined. Other relevant work is (Toussaint et al., 2010).

Publications

M. Gienger, M. Toussaint, N. Jetchev, A. Bendig, and C. Goerick. Optimization of fluent approach and grasp motions. In *8th IEEE-RAS International Conference on Humanoid Robots*, 2008.

N. Jetchev and M. Toussaint. Trajectory prediction: Learning to map situations to robot trajectories. In *26th Int. Conf. on Machine Learning (ICML)*, pages 449–456, 2009.

N. Jetchev and M. Toussaint. Trajectory prediction in cluttered voxel environments. In *Int. Conf. on Robotics and Automation (ICRA)*, pages 2523–2528, 2010.

N. Jetchev and M. Toussaint. Task space retrieval using inverse feedback control. In *28th Int. Conf. on Machine Learning (ICML)*, pages 449–456, 2011.

M. Toussaint, N. Plath, T. Lang, and N. Jetchev. Integrated motor control, planning, grasping and high-level reasoning in a blocks world using probabilistic inference. In *ICRA*, pages 385–391, 2010.

Submissions

N. Jetchev and M. Toussaint. Fast motion planning from experience: Trajectory prediction for speeding up movement generation, 2011. Submitted.

Chapter 2

Background: Motion Generation and Learning

2.1 Kinematics of Articulated Robot Motion

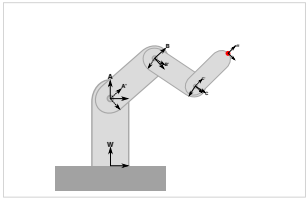
A robot consists of a set of joint axes connecting rigid body links, see Figures 2.1(a) and 2.1(b). We denote by $q = \{q_1, q_2, ...q_n\} \in \mathbb{R}^n$ the vector of the robot joint angles. This is also known as the robot configuration space, and changing the values of these angles leads to the robot moving. A kinematic map ϕ would use the robot geometry (fixed and unchanging) and the current joint angles q (variable) to find the value of the position of some robot endeffector $\phi : q \mapsto y \in \mathbb{R}^d$. The matrix of partial derivatives of a kinematic map is called the Jacobian matrix $\mathcal{J} = \dfrac{\partial \phi}{\partial q} \in \mathbb{R}^{d \times n}$. If $d = 1$ we will call $\mathcal{J}$ the gradient instead of the Jacobian. The interpretation of the Jacobian is that it indicates how much the joint angles should change in order to change the value of this kinematic map.

Both the kinematic map and its Jacobian can be calculated by simple geometrical operations chaining rotations and translations along the robot joints and links. We skip all further details about this, and throughout this thesis we assume that we have some module (simulator) that can calculate these kinematic maps whenever we need them. We refer the reader to Craig (1989); Siciliano and Khatib (2008); Toussaint (2011) for further details on how kinematic maps can be calculated using the robot geometry.

2.1.1 Motion Features and Task Spaces

Motion features will be used as a general term describing any information we can get from the simulator. These include kinematic maps of the robot, but also any other information that is available in a world filled with objects. Such information can come from external sensors (3D cameras, lasers) and would typically localize external objects, not part of the body of our robot. Such external features can not be controlled by the robot joint angles, but provide useful context information for certain workspaces and tasks. For example , if a robot needs to hit a ball (external object), the information of the ball motion would

(a) The joints as they look on real Schunk LWA3 hardware with 7 arm joints connecting the arm links.



(b) A simplified scheme of 3 different joints, their frames and 4 body links. Image taken from Toussaint (2011)

Figure 2.1: Articulated robot scheme: connected joints and rigid body links.

be a useful motion feature. The robot can not directly influence the motion of the ball, but it is still important for the calculation of the robot's own motion in order to hit the ball.

We will use the word *task space* specifically for a motion feature describing the motion of the robot itself. Task spaces can be controlled by changing the robot joint angles, as determined by the robot kinematics and appropriate kinematic maps. Here we give some basic examples of task spaces we can use if we know the robot kinematics and joint angles:

- The position (x, y, z coordinates) of body i in the world frame is $p_i(q) \in \mathbb{R}^3$.
- The relative position of body i in the frame of body j is $p_{i,j}(q) \in \mathbb{R}^3$.
- We can calculate from each position a new feature - its norm $d_{i,j} = \|p_{i,j}\|$.
- Analogously to $d_{i,j}$, one can use mathematical transformations of some task space to define a new task space. It is also straightforward to calculate a Jacobian matrix for such a derived task space.
- The collision costs are defined as some function of the closest distance between rigid bodies, usually the robot body and some obstacle in the environment.
- The joint "comfort" variable would penalize the joint angles if they go away from some predefined values.

2.2 Motion Models for Forward Control - Inverse Kinematics

In the previous section we gave examples of the various task spaces that can be used in an articulated robot system. Suppose we have multiple criteria of desired robot positions in different task spaces, coded in an objective function that is minimal when the robot fulfills them. *Inverse Kinematics* (IK), one of the most fundamental robotic methods for control, can be used to calculate motion satisfying these task space criteria. This

algorithm is essential for the understanding of the novel methods we will present later in this thesis, so we will write down here a variation of IK taken from Toussaint (2011), using notation similar to Craig (1989); Siciliano and Khatib (2008).

- First, $\|x\|^2 = x^T x$ is the standard squared euclidean vector norm of $x \in \mathbb{R}^k$, or L_2 norm. We can also define a metric parametrized by a precision matrix $C \in \mathbb{R}^{k \times k}$ and we write $\|x\|^2_C = x^T C x$.

- For each task space for $i = 1...M$ we have:
  – Kinematic mapping $y_i = \phi_i(q) \in \mathbb{R}^{d_i}$
  – Jacobian J_i measured at q_t such that using linearization $\delta y_i = J_i \delta q$. [1]
  – Current value $y_{i,t} = \phi_i(q_t)$.
  – Desired value y^*_i.
  – Metrics in task space expressed via a matrix $C_i \in \mathbb{R}^{d_i \times d_i}$.
  – The metric in joint space $W \in \mathbb{R}^{n \times n}$ for joint space movements.[2]

- Each criteria contributes a term to the objective function

$$\begin{aligned} C(q, q_t) \; &= \|q - q_t\|^2_W \\ &+ \|\phi_1(q) - y^*_1\|^2_{C_1} \\ &+ \cdots \\ &+ \|\phi_M(q) - y^*_1\|^2_{C_M} \end{aligned}$$

- The optimum of this quadratic expression is the IK equation:

$$q^* = q_t + \Big[\sum_{i=1}^{m} J_i^T C_i J + W\Big]^{-1} \Big[\sum_{i=1}^{m} J_i^T C_i (y^*_i - y_{i,t})\Big] \tag{2.1}$$

The cost $C(q, q_t)$ is small when the robot makes steps close to its current joint positions (because of the term $\|q - q_t\|^2_W$) and simultaneously fulfills all task space criteria. IK allows us to control in all these multiple task spaces simultaneously, and we can think of the control method as a puppeteer pulling different strings on a puppet, each string representing some task and the strength of the pull the task importance.

If one makes iteratively steps using IK at some fixed time resolution τ one is effectively controlling the endeffector velocity $\dot{y} = \frac{\delta y}{\tau}$. Generating a sequence of steps $\{q_t\}$ starting from q_0 with IK would be called *motion rate control*:

$$q_{t+1} = \underset{q}{\text{argmin}}\; C(q, q_t) \tag{2.2}$$

[1] δy refers to a change in task space value, and δq to a change in joint space.
[2] To simplify notation we omit the joint metric W later in this thesis.

Potential Based Control

Another classical control model is potential based control, where the robot is attracted or repelled by different potentials. This can be thought as an instance of Motion Rate Control with specific choices of task variables, e.g. a task term to go near a target and at task term to go away from any obstacle.

2.3 Motion Models for Planning

In this section we will introduce motion planning and present various algorithms used in robotics.

2.3.1 Robot Motion Planning: a Basic Model

For many robot tasks it is essential to plan not only a step towards the goal, but a whole trajectory, i.e. a motion plan with start and endpoint in time. Motion planning means generating motion with multiple steps over some time horizon in order to solve some task (LaValle 2006). Let us describe the robot configuration at time t as $q_t \in \mathbb{R}^N$, the joint posture vector. We define $\boldsymbol{q} = (q_0, .., q_T)$ as a movement trajectory with time horizon of T steps. In a given situation x, i.e., for a given initial posture q_0 and the positions of obstacle and target objects in this problem instance a typical motion planning problem is to compute a trajectory which fulfills different criteria, e.g. an energy efficient movement not colliding with obstacles. We formulate this as an optimization problem by defining a cost function

$$C(x, \boldsymbol{q}) = \sum_{t=1}^{T} g_t(q_t) + h_t(q_t, q_{t-1}) \; . \tag{2.3}$$

that characterizes the quality of the joint trajectory in the given situation and task constraints. We will specify such cost functions explicitly in our experiments section. Generally, g will account for task targets and collision avoidance, and h for control costs, similar to the terms involved in IK. A trajectory optimization algorithm (like the ones we mentioned in Section 1.1.1) essentially tries to map a situation x to a trajectory $\boldsymbol{q}$ which is optimal,

$$x \mapsto \boldsymbol{q}^* = \underset{\boldsymbol{q}}{\operatorname{argmin}} \, C(x, \boldsymbol{q}) \; . \tag{2.4}$$

For this we assume to have access to $C(x, \boldsymbol{q})$ and local (linear or quadratic) approximations of $C(x, \boldsymbol{q})$ as provided by a simulator, i.e., we can numerically evaluate $C(x, \boldsymbol{q})$ for given x and $\boldsymbol{q}$ but we have no analytic model. To arrive at the optimal trajectory $\boldsymbol{q}^*$ (or one with a very low cost C), most local optimizers start from an initial trajectory $\tilde{\boldsymbol{q}}$ and then improve it, using the partial derivatives (and Hessian matrix) of each cost term with respect to the joint states. We call $\mathcal{O}$ the local optimization operator and write $\boldsymbol{q}^* = \mathcal{O}_x(\tilde{\boldsymbol{q}})$ when we optimize in a specific situation x.

Optimizing the trajectory cost C is a challenging high-dimensional nonlinear problem. Using a direct approach and making gradient descent in joint space is one option to approach it. Optimization and control methods designed specifically to optimize motion trajectories like iLQG (Todorov and Li 2005) and AICO (Toussaint 2009) seem to work better (in terms of convergence speed and robustness), and we will be using these in our experiments. However, many of the movement optimization methods are sensitive to initial conditions and their performance depends crucially on it. For example, initial paths going straight through multiple obstacles are quite difficult to improve on, since the collision gradients provide confusing information and try to jump out of collision in different conflicting directions. The next section will present a different class of planning methods that specifically focus on finding collision free paths.

2.3.2 Sampling Based Planning

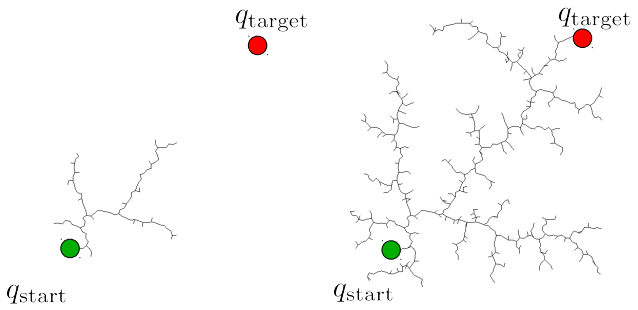
The local optimization methods we mentioned in the previous section differ qualitatively from path planners using randomized search: finding obstacle free paths in the configuration and work space of the robot via random sampling. Examples are Rapidly-exploring Random Trees (RRT) LaValle (2006) and probabilistic road maps Kavraki et al. (1995). The main idea of this class of methods is to use random sampling to build networks of feasible configuration nodes. These methods aim at global feasibility and are well suited to solve complex motion puzzles in cluttered scenes, which could not be tackled using only local optimizers.

Here we present a possible implementation of the RRT algorithm (as defined in Toussaint 2011): the robot starts in q_{start} and should go to q_{goal}. It assumes that some routine (a local planner or IK) is available to calculate whether each edge $\{(q_{\text{near}}, q_{\text{new}})\}$ can be actually followed by the robot in a collision free way.

Algorithm 1 Rapidly-exploring Random Trees Algorithm

Require: q_{start}, q_{goal}, number n of nodes, stepsize α, β
Ensure: tree $T = (V, E)$
1: initialize $V = \{q_{\text{start}}\}$, $E = \emptyset$
2: **for** $i = 0 : n$ **do**
3: **if** rand$(0, 1) < \beta$ **then** $q_{\text{target}} \leftarrow q_{\text{goal}}$
4: **else** $q_{\text{target}} \leftarrow$ random sample from Q
5: $q_{\text{near}} \leftarrow$ nearest neighbor of q_{target} in V
6: $q_{\text{new}} \leftarrow q_{\text{near}} + \frac{\alpha}{|q_{\text{target}} - q_{\text{near}}|}(q_{\text{target}} - q_{\text{near}})$
7: **if** $q_{\text{new}} \in Q_{\text{free}}$ **then** $V \leftarrow V \cup \{q_{\text{new}}\}$, $E \leftarrow E \cup \{(q_{\text{near}}, q_{\text{new}})\}$
8: **end for**

Once a feasible path to the target is found (path from root to leaf in the tree) it is often jagged because of the random sampling, see Figure 2.2. RRT paths require additional optimization by a local planner to get a (dynamically) good trajectory that can be executed on a robot with additional motion optimality constraints coded in some cost



(a) After $n = 100$ samples. (b) After $n = 400$ samples.

Figure 2.2: A scheme of an RRT planner in 2D configuration space: starting from q_{start} the planner builds a feasible path towards the goal q_{target}. Image taken from Toussaint (2011).

function. Thus, sampling based planning in conjunction with local planning is a powerful method, but at a disadvantage because of the long time to process information before any motion is generated.

2.3.3 On Planning and Control

In Figure 2.3 we illustrate graphically IK control, local planners (trajectory optimization) and RRT planners. A fundamental difference between reactive control (e.g. IK) and planning in general is that reactive control starts moving immediately, while planning requires some time to prepare a plan as output. This reactive and immediate aspect can be useful for quickly generating simple motions, but has the drawback that the motions are more prone to failure. Planners, on the other side, can construct motions of much better quality and reason about the future steps of the robot and their effects, e.g. anticipating whether a movement leads to a dead-end. The local planners create smooth motions that have multiple desired properties, but because their optimization can have local nature it can also get stuck occasionally, e.g. if the obstacles have very complex geometry. RRTs have more global character and can (provably) reach any solution given enough time, but their motions are not smooth at all. However, there is the trade-off

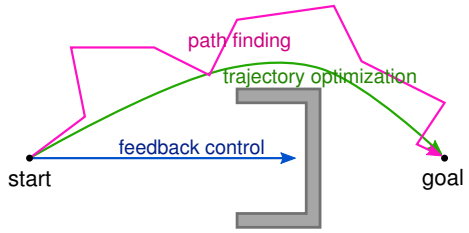


Figure 2.3: A scheme of 3 basic planning and control methods (IK control, local planning and RRT). Image taken from Toussaint (2011).

that planners need some time to plan before outputting any motion. For time-critical applications requiring immediate reaction this can be a drawback.

Note that all planning and control algorithms we presented here make no use of experience so far. Our novel algorithms TP and TRIC will alleviate some of these issues: smart prediction can shorten the planning time of a planner, and well constructed task space can make a reactive controller more efficient globally.

2.4 Machine Learning and Imitation Learning Methods

Here we present an overview of methods for motion generation based on observing and repeating sample motion trajectories. Unlike the planning and control methods we presented so far, imitation learning methods make no assumption that a cost function specifying desired motions exists. Rather, the challenge is to learn approximate models (e.g. with machine learning) of this cost using the available data that can be used to generate motion.

2.4.1 Direct Policy Learning

Direct Policy Learning (DPL) is one of the fundamental approaches for imitation learning, see Pomerleau (1991); Argall et al. (2009). Sometimes it is also called "behavior cloning", because it is a straightforward approach that tries to repeat observed motions.

A standard way to describe a robot trajectory is $\{x_t, u_t\}_{t=0}^{T}$, where x_t represents the robot state at time t and u_t is the control signal, e.g. the rate of change $\dot{x}_t$. DPL tries to find a policy $\pi : x_t \mapsto u_t$ from these observations. Different assumptions can be made for the choice of x,u and π (Calinon and Billard 2007), with refinements like data transformations and active learning. Given a parameterization of the policy, DPL essentially corresponds to a regression problem, e.g. with loss:

$$E_{\text{dpl}} = \sum_{t=0}^{T} \|\pi(x_t) - u_t\|^2 \tag{2.5}$$

where $\|.\|^2$ denotes the squared L_2 norm. Minimizing E_{dpl} finds a policy close in the least squares sense to the demonstrations. The above loss can be extended to multiple demonstration trajectories by averaging over them.

Howard et al. (2009) introduces an interesting alternative loss for DPL:

$$E_{\text{inc}} = \sum_{t=0}^{T} (\|u_t\| - \pi(x_t)^T u_t / \|u_t\|)^2 \tag{2.6}$$

This loss penalizes the discrepancy between the projection of the policy $\pi(x_t)$ on u_t and the true control u_t. Howard et al. (2009) show that in some problem domains this loss leads to better behavior than the standard least squares loss.

When the state and control spaces are high dimensional DPL has a disadvantage: generalization is an issue and would essentially require the data to cover all possible situations. The approach we develop in Chapter 5 will aim to improve generalization by extracting the relevant task features from data and by finding an underlying structure of multiple demonstrated trajectories.

2.4.2 Markov Decision Process and Reinforcement Learning

A Markov Decision Process (MDP) is a graphical model involving world states s (e.g. robot position) and actions a (e.g. go left). It is a popular formalism for multiple learning problems, including Reinforcement Learning (RL), see e.g. Russell and Norvig (2009). The MDP is defined by the following probabilities, taken for reference from Toussaint (2011):

- world's initial state distribution $P(s_0)$
- world's transition probabilities $P(s_{t+1} \mid a_t, s_t)$
- world's reward probabilities $P(r_t \mid a_t, s_t)$ and $R(a, s) := \mathrm{E}\{r|a, s\}$
- agent's *policy* $\pi(a_t \mid s_t) = P(a_0|s_0; \pi)$

The **value** (*expected* discounted return) of policy π when started in state s with discounting factor $\gamma \in [0, 1]$ is defined as:

$$V^\pi(s) = \mathrm{E}_\pi \left\{ r_0 + \gamma r_1 + \gamma^2 r_2 + \cdots \mid s_0 = s \right\} \tag{2.7}$$

One way to do reinforcement learning in a MDP is to iterate the Bellman optimality equation until convergence of the value function - Value Iteration algorithm (Bellman 1957):

$$V^*(s) = \max_a \left[R(a, s) + \gamma \sum_{s'} P(s' \mid a, s)\, V^*(s') \right]$$

The optimal policy given the optimal value function is simply the policy maximizing the immediate reward and expected future rewards:

$$\pi^*(s) = \operatorname*{argmax}_a \left[R(a, s) + \gamma \sum_{s'} P(s' \mid a, s)\, V^*(s') \right]$$

The value of a state $V(s)$ is a more global indicator of desired states than the immediate reward of an action $R(a, s)$. The values $V(s)$ provide a gradient towards desired states - going in direction of increasing $V(s)$ is the desired behavior of the robot system. [3]

In Chapter 5 we will learn value functions in order to get effective motion policies.

[3] Usually in RL one has *rewards* that are desired to be large, while in robotics people uses *costs* which are desired to be low. The algorithms remain equivalent up to a change of sign and maximization/minimization.

2.4.3 Inverse Optimal Control

RL or planning in general (e.g. within a MDP framework) tries to generate motions maximizing some reward. Learning (e.g. with Value Iteration) requires constant feedback in the form of rewards for the states and action of the agent. However, in many tasks the reward is not analytically defined and there is no way to access it from the environment accurately. It is then up to the human expert to design a reward leading the robot to the desired behavior. There is an algorithm that can effectively learn the desired behavior and a policy for it just by observing example motions. Inverse Optimal Control (IOC), also known as Inverse Reinforcement Learning (IRL), aims to limit the reward feedback requirement and human expertise required to design behavior for a task. IOC learns a proper reward function only on the basis of data, see Ratliff et al. (2006). Let's assume that policies π give rise to expected feature counts $\mu(\pi)$ of feature vectors ϕ: i.e. what feature we will see if this policy is executed. A weight vector w such that the behavior demonstrated by the expert π^* has higher expected reward (negative costs) than any other policy is learned by the minimizing a loss:

$$\min \ |w|^2 \tag{2.8}$$

$$s.t. \ \forall \pi \ \ w^T \mu(\pi^*) > w^T \mu(\pi) + \mathcal{L}(\pi^*, \pi) \tag{2.9}$$

The term $w^T \mu(\pi)$ defines an expected reward, linear in the features. The scalable margin $\mathcal{L}$ penalizes those policies that deviate more from the optimal behavior of π^*. The above loss can be minimized with a max margin formulation. Efficient methods are required to find the π that violates the constraints the most and add it as new constraint. Once the reward model is learned, another module is required to generate motions maximizing the reward, e.g. Ratliff et al. (2006) uses an A^* planner to find a path to a target with minimal costs.[4]

Learning a policy based on estimated costs is much more flexible than DPL, and a simple cost function can lead to complex optimal policies. IOC can often generalize well to new situations, because states with low costs create a task manifold, a whole space of desired robot positions good for the task. In some domains it is much easier to learn a mapping from state to cost than to learn a mapping from state to action. The latter is a more complex and higher dimensional problem, especially when considering actions in high dimensional continuous spaces such as robot control. In Chapter 5 we will present our method TRIC which is inspired by IOC, in that it assumes that the teacher was optimal with respect to some latent criteria, which can be then recovered from motion demonstration data.

2.4.4 Discriminative Learning

Discriminative learning provides a common framework for many learning problems, including structured output regression. Popular approaches include large margin models (Tsochantaridis et al. 2005) and energy based models using neural networks (LeCun

[4] A^* is related to the Dijkstra graph shortest paths algorithm.

et al. 2006). Data is given in the form of pairs of input and output values $\{x_i, y_i\}$. As in standard discriminative approaches (e.g., structured output learning), the energy or cost $f(x_i, y_i; w)$ provides a discriminative function such that the true output should get the lowest energy from the model f:

$$y_i = \underset{y \in \mathcal{Y}}{\text{argmin}}\ f(x_i, y) \tag{2.10}$$

Training the parameter vector w of the model f is done by minimizing a loss over the dataset. The loss should have the property that f is penalized whenever the true answer y_i has higher energy than the false answer with lowest energy which is at least distance r away:

$$\tilde{y}_i = \underset{y \in \mathcal{Y}: \|y - y_i\| > r}{\text{argmin}}\ f(x_i, y) \tag{2.11}$$

Finding the most offending answer $\tilde{y}_i$ is very often a complicated inference problem in itself.

Instead of the common hinge loss $L_{hinge}(x_i, y_i) = \max(0, m + f(x_i, y_i) - f(x_i, \tilde{y}_i))$ we will be using the log loss:

$$L_{log}(x_i, y_i) = \log(1 + e^{f(x_i, y_i) - f(x_i, \tilde{y}_i)}) \tag{2.12}$$

which is a soft form of L_{hinge} with infinite margin (LeCun et al. 2006).

In Chapter 5 our method TRIC will use some ideas from discriminative learning for the purpose of learning a value function with certain discriminative properties from motion demonstrations .

Chapter 3

Trajectory Prediction: Mapping Situations to Motions

The first approach for learning representations which we present in this thesis is *Trajectory Prediction* (TP). The insight of this method is to learn a mapping from a descriptor of the situation to a whole motion trajectory in a task-appropriate way, resulting in a novel approach to speeding-up motion planning and analysis of the structure of situations.

3.1 Overview of TP

Trajectory planning and optimization is a fundamental problem in articulated robotics. Algorithms used typically for this problem compute optimal trajectories from scratch in a new situation, without exploiting similarity to previous problems. In effect, extensive data is accumulated containing situations together with the respective optimized trajectories – but this data is in practice hardly exploited. This chapter of the thesis describes a novel method to learn from such data and speed up motion generation, *Trajectory Prediction*, first described in our previous works Jetchev and Toussaint (2009, 2010).

The main idea is to use demonstrated optimal motion trajectories to predict appropriate trajectories for novel situations. These can be used to initialize and thereby drastically speed-up subsequent optimization of robotic movements and improve the convergence behavior of a conventional motion optimizer. Our approach has two essential components. First, to generalize from previous situations to new ones we need an appropriate situation descriptor – we construct features for such descriptors and use a sparse regularized feature selection approach to find well-generalizing features of situations. Second, the transfer of previously optimized trajectories to a new situation should not be made in joint angle space – we propose a more efficient task space transfer of old trajectories to new situations.

We present extensive results in simulation to illustrate the benefits of the new method, and demonstrate it also with real robot hardware. Our experiments with a reaching and obstacle avoidance task, and an object grasping task, show that we can predict good

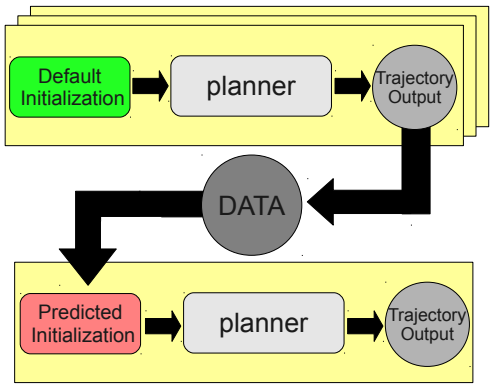


Figure 3.1: A diagram illustrating TP: we gather trajectory data from multiple runs of the motion planners; this data can then be used to predict a smarter initialization speeding-up planner performance in a novel situation.

motion trajectories in new situations for which the refinement is much faster than an optimization from scratch.

The main contributions of this thesis in the area of trajectory prediction can be summarized as follows:

- the TP method for speeding up planning by learning a predictive model for cost-sensitive classification of motion trajectories likely to be good planner initializations depending on the situation
- the definition and refinement of representations that allow accurate mapping of situation to movement
- the notion of IK Transfer in task space, allowing robust generalization of the predicted movements between different situations
- quantitative results in three different motion planning tasks showing how TP can speed up motion planning

In Section 3.2 we will give an overview of TP and how it is coupled with motion planning. Afterwards we will proceed with Section 3.3 where we examine what representations of states allow efficient generalization of movements between situations. In Section 3.4 we will present the Inverse Kinematics (IK) Transfer operator we use, and discuss what task space representations are appropriate for such transfer. The way we learn a policy used for predicting situation-appropriate trajectories will be explained in Section 3.5. Further discussion in Section 3.6 will elaborate connections between TP and imitation learning, and discuss what motivated taking a different approach for our novel TP framework. Afterwards, in the experimental Section 3.7 we will show our results for several simulated robot motion planning scenarios. We finish the chapter with our conclusions in Section 3.8.

3.2 Planning Motion and Predicting Motion

We assume that the desired behavior of the robot is to generate a motion trajectory good for some specified task. As mentioned in the background section 2.3.1, such a desired trajectory can be calculated by a planner module minimizing a cost function. One can think of the behavior of such a planner (and some heuristic for initialization) as a policy mapping a situation x to a joint trajectory $\boldsymbol{q}$. We propose to use experience in the form of demonstrated optimal trajectories in different situations as initialization for local planners, resulting in a better movement policy. This main idea behind TP is illustrated in the diagram in Figure 3.1. Below we give a summary of the assumptions and premises that need to hold in order to use TP:

Premises for the TP framework:

- a **cost function** that characterizes how good a trajectory is for a **task**
- a **planning algorithm** is available that can find low-cost trajectories
- the task is to be executed in different **situations** sampled from an underlying **generating distribution**
  "the objects in the world, their count, size and positions, have a certain pattern"
- **data** consisting of situation descriptors and motions good for the task
  "observe the robot motion and observe all information available from sensors of the world"

Given these assumptions, TP can **accomplish** the following:

- use the gathered training data to **predict** motion trajectories in any new test situation and use them as **planner initialization**
  "improve planning in situations sampled from the underlying distribution"
- selection of **sparse** situation features

The above assumptions necessary for TP are reasonable for many robot tasks, where we have a description of the task to be performed (a cost function usually) and expect to repeat this task in a structured environment. It is also reasonable to have some planner for such tasks, and it is always an advantage to speed-up motion generation. The assumption that the training situations have similar optimal trajectories as the test trajectories is related to the classical *stationarity assumption* in learning theory, see Russell and Norvig (2009). This is the assumption that examples of input-output pairs seen in the past have the same distribution as the input-output pairs encountered in the future, here situation-motion pairs.

This section will proceed by formalizing TP.

3.2.1 Trajectory Prediction: Overview of the Algorithm

In this section we first define the trajectory prediction problem in general terms and outline how we break down the problem in three steps: (i) finding appropriate task space descriptors, (ii) transfer of motion prototypes to new situations, and (iii) learning a predictive model of which motion prototype is appropriate to be transferred to a new situation.

The goal of TP is to learn an approximate model of the mapping (2.4), i.e. from starting robot position to a final configuration, from a data set of previously optimized trajectories. The dataset D comprises pairs of situations and optimized trajectories,

$$D = \{(x_i, \boldsymbol{q}_i)_{i=1}^{d}\} \;, \quad \boldsymbol{q}_i \approx \underset{\boldsymbol{q}}{\operatorname{argmin}}\, C(x_i, \boldsymbol{q}) \;. \tag{3.1}$$

The full sequence involved in TP is the following:

$$x \rightarrow \hat{i} \rightarrow \mathcal{T}_{xx_{\hat{i}}} \boldsymbol{q}_{\hat{i}} \rightarrow \mathcal{O}_x \mathcal{T}_{x_{\hat{i}} x} \boldsymbol{q}_{\hat{i}} = \boldsymbol{q}^* \tag{3.2}$$

TP takes as input an appropriately represented situation descriptor x, see Section 3.3. We then predict the index $\hat{i}$ of a motion from D to be executed and transfer it with the operator $\mathcal{T}$ from situation $x_{\hat{i}}$ to x, described in Section 3.4. We can view the subsequence

$$f : x \rightarrow \mathcal{T}_{x_{\hat{i}} x} \boldsymbol{q}_{\hat{i}} \tag{3.3}$$

as the policy mapping situation to motion, and will explain it in Section 3.5. Finally, the above TP sequence ends with applying the planning operator $\mathcal{O}_x$, which takes an input joint trajectory initialization and modifies it to minimize its cost, see section 2.3.1. Prediction without any subsequent optimization would correspond to pure imitation, and our method is not designed with such aim. TP is inherently coupled with a planner that minimizes the cost function C, so the prediction policy is designed to speed-up such a planner.

In this formulation of TP we are using directly the trajectories $\boldsymbol{q}$ for motion represenation. An alternative can be to use some parametrized representations of the spatio-temporal aspects of the motions, e.g. splines or compositions of movement primitives. For example, Ilg et al. (2004) extract basic movement primitives from example trajectories, and use linear combinations of these basic primitives to efficiently represent motions of different styles.

As an aside, the TP problem setup generally reminds of structured output regression (Tsochantaridis et al. 2005, see also Section 2.4.4). However, in a structured output scenario one learns a discriminative function $C(x, \boldsymbol{q})$ for which $\underset{\boldsymbol{q}}{\operatorname{argmin}}\, C(x, \boldsymbol{q})$ can efficiently be computed, e.g. by inference methods. Our problem is quite the opposite: we assume $\underset{\boldsymbol{q}}{\operatorname{argmin}}\, C(x, \boldsymbol{q})$ is very expensive to evaluate and thus learn from a data set of previously optimized solutions. A possibility to bring both problems together is to devise approximate, efficiently computable structured models of trajectories and learn the approximate mapping in a structured regression framework. But this is left to future research.

In the next sections we will continue with detailed description of the elements of TP.

3.3 Situation Representations and Descriptor

A typical scenario for articulated motion generation is a workspace filled with objects and a robot. A situation (or problem instance) is fully specified by the initial robot posture q_0 and the positions of obstacles and targets in this problem instance. There are a lot of possible features we can construct to capture relevant situation information. For instance, positions of obstacles could be given relative to some coordinate system in the frame of some other object in the scene. We should expect that our ability to generalize to new situations crucially depends on the representations we use to describe situations.

We present two different approaches for modeling a descriptor x, appropriate for different types of workspace situations.[1] This section will proceed by describing these two models.

3.3.1 General Geometric Descriptor

Our first approach is to define a very high-dimensional and redundant situation descriptor which includes distances and relative positions w.r.t. many different frames of reference. Training the predictive function then includes selecting the relevant features. Assume we have a set of b different 3D objects (i.e. landmarks) in the scene which might be relevant for motion generation: $A = \{a_j\}_{j=1}^{b}$. Each a_j has both a 3D position and a rotational frame, usually from the body it is attached to. We create features by examining the geometric relations between pairs of such objects. For b such landmarks we have $\hat{b} = b(b-1)$ such pairs. Because of the number of pairs being quadratic in the number of landmarks, this descriptor is appropriate in workspace situations where we have a small fixed set of important rigid bodies interacting. A typical example would be a workspace with an articulated robot with an endeffector and other body parts as landmarks, and a single object with simple geometry as manipulation target, see Figure 3.2 for an illustration.

For each pair $j = (j_1, j_2) \in \{1, ..., \hat{b}\}$ we measure the 3D relative difference vector between a_{j_1} and a_{j_2} in the frame of a_{j_2} as $p_j = (p_j^x, p_j^y, p_j^z)$. The norm of p_j is another useful feature we use: $d_j = \|p_j\|$. We also define the azimuths of the three axes as $\psi_j = \{\arccos(p_j^x/d_j), \arccos(p_j^y/d_j), \arccos(p_j^z/d_j)\}$.

We gather this basic geometric information in the 7 dimensional vector (p_j, d_j, ψ_j). The final descriptor x comprises all these local pairwise vectors concatenated:

$$x = (p_1, d_1, \psi_1, ..., p_{\hat{b}}, d_{\hat{b}}, \psi_{\hat{b}}) \in \mathbb{R}^{7\hat{b}} \tag{3.4}$$

Note that the set of landmarks is not permutation invariant with respect to the indexing of landmarks $\{a_j\}_{j=1}^{b}$. If we have several identical objects and landmarks on them, the order of the landmarks in set A will influence the descriptor shape. Using relational representations can be a potential way to deal with this issue. However, the

[1] Note that we use x as both the situation descriptor and the situation itself in the cost $C(x, q)$. The reason is that each cost $C(x, q)$ is calculated always for a specific situation x with certain positions of objects in the world that also determine the information captured in the descriptor.

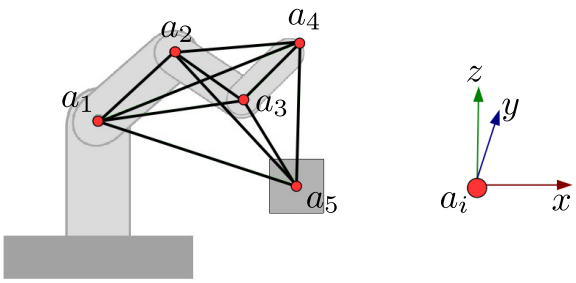


Figure 3.2: The geometrical descriptor captures the distances between pairs of landmarks a_j. A simple illustration with a_1, a_2, a_3 and a_4 as landmarks on the robot, and a_5 as a landmark on an object in the situation. Each landmark has a position and rotational frame.

lack of permutation invariance is not problematic when having enough data and not too many landmarks. More complex geometric relations between landmarks can be used for descriptor features, but the choice of (3.4) turned out to be sufficient in our experiments. Given such a descriptor we can use a feature selection technique to infer from the data which of these dimensions are best for trajectory prediction in new situations. In the experimental Section 3.7 we will show how extracting a sparse representation from this redundant description provides an interesting explanation of the important factors in a situation giving rise to different motions.

3.3.2 Voxel Descriptor

The approach to use the distances between landmark centers as features is appropriate for situations with few obstacles with simple geometries, but it can have scaling issues when more objects are present. The count of pairwise distance features increases quadratically with respect to the number of objects and there is no permutation invariance, so the descriptor from Section 3.3.1 can be impractical. Another drawback of the first descriptor is that it needs to use objects of fixed sizes. If the training data containing the coordinates of an obstacle and a target object of one size, than no generalization is possible to objects of other sizes in a new situation. If the object sizes were added as features in the descriptor there would be more generalization capabilities, but often in practical robot applications one does not have accurate geometric models of the obstacles encountered in the workspace.

We present an extension appropriate for cluttered scenes. It utilizes a descriptor where obstacles are modeled from point clouds of 3D sensor data, which can handle multiple objects of various sizes easily. We call this a sensor-driven approach to trajectory prediction. Note that it is not required that the situation features be differentiable functions or controllable by the robot, we are flexible to use any information available. We assume that a sensor (LIDAR or stereovision) is available that provides information in the form of a point cloud from detected objects, which can be used in a voxel representation of a scene, see Elfes (1989); Nakhaei and Lamiraux (2008). This information representing the obstacles is crucial for the correct task execution, an assumption appro-

priate for cluttered scenes and navigation. Given a set of laser cloud points $P = \{p_i\}$, we construct a 3D grid system $V = \{v_i\}$ of voxels. Each voxel is identified with its coordinates and its occupancy probability $p(v_i) \in [0, 1]$. The procedure for calculating $p(v)$ is straightforward:

1. Loop through all available measurements p_i
2. Loop through all voxels v_j
3. If $p_i \subset v_j$ set $p(v_j) = 0.1 + 0.9p(v_j)$

The idea is that for every measurement point within some voxel bounds the occupied space probability of the voxel increases. Elfes (1989) and Nakhaei and Lamiraux (2008) use sensor models with state distributions for free, unknown and occupied voxel space, but we used only the occupied space probability model.

To better explain the voxel descriptor, we will describe how it will look concretely in our experiments. We define two such voxel grids, 15 voxels across each dimension, where each voxel is a cube with side 7cm. The first grid V^1 is centered at the center of the workspace, the second V^2 is centered on the target location. Each voxel grid V^i can be described as a vector of dimension $15^3 = 3375$ containing the values of all its cells $p(v_i)$, but such a high dimensionality is often impractical. We can compress a voxel grid to the 200 most significant dimensions using standard Principal Component Analysis (PCA), Pearson (1901). We have the following grid descriptor in vector form:

$$\hat{V} = \bar{V}^T P \in \mathbb{R}^{200} \tag{3.5}$$

where $P \in \mathbb{R}^{3375\times200}$ is the projection matrix calculated by PCA and $\bar{V}$ is a mean-centered voxel grid. By taking only the columns of P with highest variance one can get a variance preserving compression of the voxel grids.

The final situation descriptor is

$$x = \{d, \hat{V}^1, \hat{V}^2\} \in \mathbb{R}^{413} \tag{3.6}$$

The terms $\hat{V}^1$ and $\hat{V}^2$ are the two PCA compressed descriptors of the two voxel grids V^1 and V^2, and $d \in \mathbb{R}^{13}$ contains additional scene information, the initial 7D robot arm joint position, the 3D endeffector position and 3D target position.

Voxel PCA Components as Characteristic Terrain Features

The PCA components can be interpreted as responses to characteristic terrains of the voxel grid. The projection coefficients are the eigenvectors P_i, columns of the PCA projection matrix P. Each entry P_{ji} of the vector P_i corresponds to one voxel cell v_j. The value of the ith principal component (i.e. situation feature after compression) is $\hat{V}_i = \sum_j P_{ji}\bar{v}_j$. Feature $\hat{V}_i$ is large when the occupied voxels correspond to positive P_{ji} and the free space voxels have negative P_{ji}.[2] In Figure 3.3 we visualize 6 of the PCA

[2]PCA uses centered voxel data: free voxels are negative and occupied voxels are positive numbers.

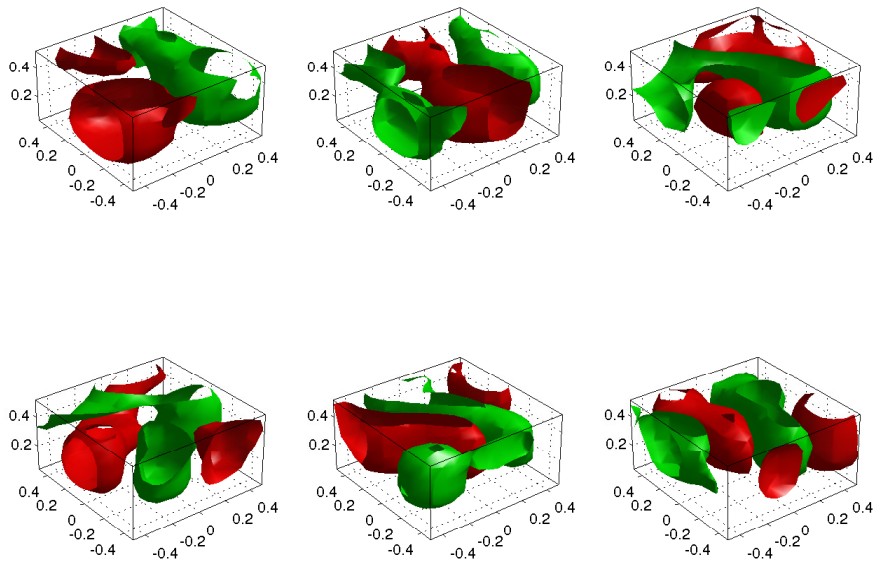


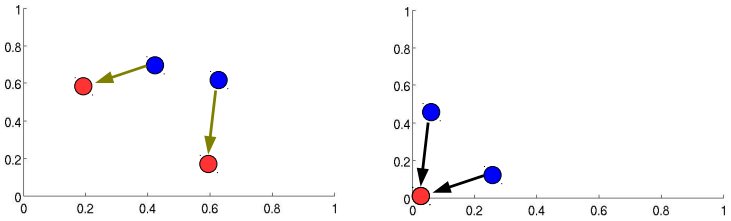
Figure 3.3: The PCA eigenvectors as indicators of characteristic terrain. Volumetric plots of the 6 PCA eigenvectors P_i with greatest eigenvalue. The red isosurface ($P_{ji} = \mu_1$) shows the boundaries of the occupied volume, and the green isosurface ($P_{ji} = \mu_2$) the free space. The constants $\mu_1 > 0 > \mu_2$ are chosen for a visualization purpose: the red and green volumes have 20 percent of the total voxel grid volume.

eigenvectors as a volume in 3D space using their correspondence to voxel cells. The value of the shown principal components is high when the terrain fits the pattern of free/occupied terrain indicated in the plots.

Such a PCA decomposition for voxel data is useful to discriminate between world configurations and represent general notions like whether the left or right side of the workspace is free. A lot of the detailed voxel information about the world is lost by PCA, but prediction is more efficient with lower dimensional descriptors.

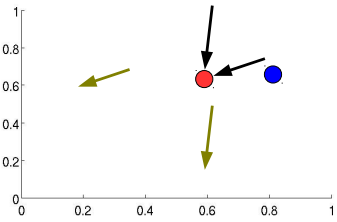
3.4 Task Space Trajectory IK Transfer

In this section we will describe the way in which we repeat and adapt a motion from the database to a random new situation. This involves three steps. First, we need to choose a representation for the output trajectory in some task space, i.e. motion features "appropriate" for the task. Second, we need to project a joint space trajectory to this task space. Finally, we need to transfer a task space trajectory back to a joint space trajectory to be used as final output to the planner.



(a) Two motions in the world frame: space Y.

(b) Two motions in a frame centered on the target: space Y_{target}.



(c) A new situation, where we attempt to reuse the previous motions. A motion from space Y_{target} is best suited to initialize a motion to the target.

Figure 3.4: A simple illustration of how a special coordinate frame represented via task space Y_{target} can help with generalization. A 2D world where the robot starts in the blue point and needs to reach the red point target. The target centered frame generalizes better than the world frame with respect to translations.

3.4.1 Motion Representation: Output Trajectory Task Space

As we mentioned in section 3.2 we will predict motion trajectories and transfer them via task space. Task space transfer has three concepts involved:

- representation of motion trajectories in a certain features, i.e. task space
- projection of joint space trajectories in the selected task space
- backprojection of task space trajectories in joint space

Note that unlike the situation descriptors we introduced in Section 3.3, the task space should contain features that are controllable by the robot joints and have Jacobians. It should also have much lower dimensionality, so that IK control in such a task space is still reasonable. Our current approach to task space selection is to test empirically task spaces that seem reasonable, and to select the space that allows the best planner initialization. This is a similar approach to Muehlig et al. (2009); Billard et al. (2004). Some obvious choices of task spaces are the joint angle space Q (a trivial space, mapped by identity from the robot configuration space) and Y, the space of world coordinates

of robot hand endeffector (mapped by the hand kinematics). However, these have the drawback of not generalizing well – a simple change in the world like translation of the position of an object would make a movement prototype in such space which was good in an old situation unfeasible for the changed situation. A reasonable choice of task space can ensure at least some degree of generalizing ability in a new situation. For example, we would also consider the task space Y_{target} of coordinates starting in a world frame centered on the target, and Y_{obst}, a world frame starting in the center of the largest obstacle in the scenes we examined. Such a choice of task space can help to generalize to translations in obstacle position (Berniker and Kording 2008). In Figure 3.4 we illustrate how space Y_{target} can help with generalization between situations and makes it more likely that an trajectory from one situation can be reused in another one.

3.4.2 Formalization of the Transfer Operator

We continue by writing the exact equations for IK Transfer and the projection and back-projection from task space involved in it.

The projection $\boldsymbol{y}$ of a trajectory $\boldsymbol{q}$ into task space is defined as:

$$\boldsymbol{y} = \phi_x(\boldsymbol{q}) \tag{3.7}$$

where ϕ_x is a kinematic mapping (depending on the situation x) applied to each time slice, with a task space for output. Another way to say this is that ϕ_x projects motions from joint space to a task space.

Suppose we want to transfer the joint motion $\boldsymbol{q}'$ which was optimal for situation x' to a new situation x. A task space trajectory $\boldsymbol{y} = \phi_{x'}(\boldsymbol{q}')$ needs to be backprojected to a joint space trajectory $\boldsymbol{q}$ in order to initialize the local motion planner in a new situation x, see Equation (3.2). Simply repeating the old motion with IK is likely to be problematic, e.g. motion targets and obstacles change between situations, so a naive replay of a trajectory in joint angles is likely to fail. Therefore we use IK with multiple task variables and motion rate control (see Section 2.2) to transfer motions and adapt to new situations. The next-step cost C^{IK} is defined as

$$C^{\text{IK}}(x, q, q_{t-1}, q'_t) = \underbrace{g(q)}_{\text{task cost}} + \underbrace{h(q, q_{t-1})}_{\text{small step}} + \underbrace{\|\phi_x(q) - \phi_{x'}(q'_t)\|^2}_{\text{follow } \phi_{x'}(\boldsymbol{q}')} \tag{3.8}$$

where q'_t is a step from the motion $\boldsymbol{q}'$ that is being transferred. The terms of C^{IK} are chosen so that making steps with low C^{IK} fulfill different criteria important for motion adaptation. The terms h and g influence the motion steps to have a low cost with respect to the terms of the (task-specific) cost function from Equation (2.3). Usually the term $h(q, q_{t-1}) = \|q - q_{t-1}\|^2$ is used to force smooth, energy efficient motions. The term $g(q)$ stays for additional criteria for good motions, e.g. avoiding collisions. The term $\|\phi_x(q) - \phi_{x'}(q'_t)\|^2$ is for following the task space trajectory.

We generate a motion for each time slice $t = 1...T$ by using motion rate control:

$$q_t = \phi_x^{-1}(q_{t-1}, q'_t) := \underset{q}{\text{argmin}}\ C^{\text{IK}}(x, q, q_{t-1}, q'_t) \tag{3.9}$$

The mapping $\phi_x^{-1} : (\boldsymbol{y}, q_0) \mapsto \boldsymbol{q}$ projects the whole task space trajectory back to a joint space trajectory in situation x, applying the IK operator ϕ_x^{-1} to minimize next-step cost C^{IK} iteratively for each time step t, starting from q_0.

The transfer operator is then the following function composition:

$$\mathcal{T}_{xx'}\boldsymbol{q}' = \phi_x^{-1} \circ \phi_{x'}(\boldsymbol{q}') \tag{3.10}$$

Such a transfer gives our method TP a better generalization ability: a predicted motion trajectory which by itself is not optimal for the motion task can still be followed and adapted in the new situation with IK. Concretely, in many cases IK Transfer will produce initial collision-free motions, which is important for planner convergence.

3.5 Mapping Situation to Motion

Once we have defined appropriate situation descriptors and the IK transfer method of adapting motions from one situation to another, we can proceed to describe the mapping f predicting "situation-appropriate" movements that lead to quick motion planner convergence. This is the last building block of TP we need to present before showing our experimental results.

3.5.1 Gathering Data Demonstrating Optimal Motions

The first step toward learning f is the gathering of optimal trajectories in different random situations. The dataset D comprises pairs of randomly generated situations x and trajectories $\boldsymbol{q}$, optimized offline to convergence with a local planner with default initialization:

$$D = \{(x_i, \boldsymbol{q}_i)_{i=1}^d\} \tag{3.11}$$

Allowing the local planner to run for lots of iterations (costing computational time) makes getting an optimal trajectory very likely, but failure is still possible, as our results will indicate later. For the set D we retained only good movements and discarded the failed attempts.

This set D constitutes a set of possible trajectories we will consider as output from TP. In the terms of reinforcement learning or DPL, we can make an analogy and call D an *action set*, because predicting a trajectory for initialization is the output from our predictive motion mapping (policy). Note also that D is created by saving motion examples from the robot experience, so we do not need to specify any prior action set. This is another level in which TP sets replaces human motion priors with a data driven framework.

Then we can gather data D' for the quality of the initialization using the saved trajectories in different situations, and measure how much additional refinement they can get from the planner for a limited amount of iterations. We measure this "refinement cost"

as

$$F(x, \boldsymbol{q}) = C(x, \mathcal{O}_x^j \boldsymbol{q}) \tag{3.12}$$

Here $\mathcal{O}_x^j \boldsymbol{q}$ is the trajectory found by the optimizer after j iterations, starting from initialization $\boldsymbol{q}$, and j is a constant. Such a definition of F is a heuristic to quantify the effect of initialization on convergence speed looking only at few planner iterations, which is possible because the planners we use iteratively improve the solution trajectory making small steps.

The cross-initialization dataset D' is defined as:

$$D' = \{(x_j, x_i, F(x_j, \mathcal{T}_{x_j x_i} \boldsymbol{q}_i)\} \ , \quad x_j \in D^x, (x_i, \boldsymbol{q}_i) \in D \tag{3.13}$$

That is, we evaluate the quality of initialization in situation x_j of a database movement $\boldsymbol{q}_i$ transferred from x_i, and this is the data we will use to learn a good mapping f. The set D^x has a new set of situations where we examine the cost of transferred motions from set D.

A difference between the datasets D and D' is due to the different initializations used to create them. For the set D we use the planners with default initialization (without experience of previous situations) to get optimal movements for the different situations, and retain only the successful runs with low costs after planner convergence (measured by C). In the second dataset D' we use examples of good motions from set D, and use IK transfer to adapt the trajectories(actions) for better generalization between situations. Another difference is that in D' we use the refinement cost (measured by F) for a limited amount of iterations.

On the Complexity of Gathering Data for TP

Creating D requires $d = |D|$ planner calls, each of which iterates until convergence to a local optima. We define $d_x = |D^x|$ as the number of situations in the dataset D'. Creating D' requires $d_x d$ planner calls, each of which takes j planner iterations. Obtaining such data D' can be expensive for complex tasks that are slow to be optimized. Data gathering can be made faster by running for fewer iterations j to evaluate F, but this is a trade-off because sometimes the first iterations are not indicative of the final costs of the planner after convergence. In Section 3.7 we will also test using smaller representative sets of motions (e.g. by using clustering) to select smaller subsets of D with different motion types. This would be a compression of the trajectory sets allowing to evaluate a smaller number of costs for initializations. However, even if gathering data is a slow process, this is an operation done offline and effectively split up on multiple available CPUs. Usually it is of great practical utility to invest time to gather data and train TP on it, because only the online performance of the combination of TP and a planner matters in typical robotic usage scenarios.

Here we describe the simple heuristic we developed to estimate the number of iterations required for F. For each situation x from the set of motions D optimized to

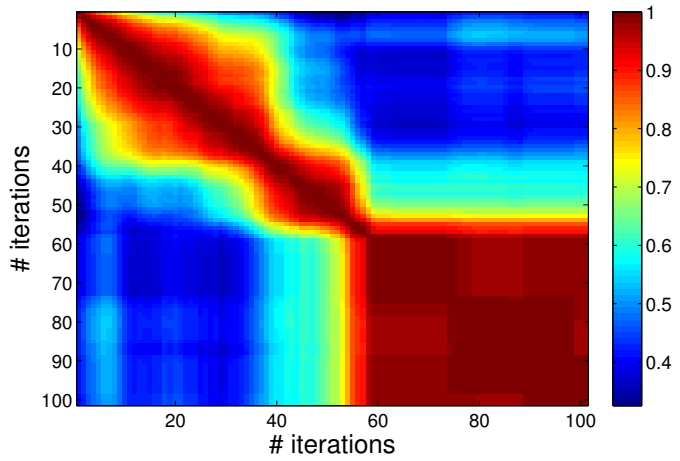


Figure 3.5: Correlation structure of planner convergence from dataset D. It allows us to visually answer the question how many iterations are necessary to estimate to what costs the planner will converge.

convergence, we can define the cost convergence vector:

$$c = (C(x, \mathcal{O}_x^1 \boldsymbol{q}), C(x, \mathcal{O}_x^2 \boldsymbol{q}), ...) \tag{3.14}$$

c consists of the costs of the trajectories found by the planner after different numbers j of iterations: $c_{[j]} = C(x, \mathcal{O}_x^j \boldsymbol{q})$. Given a set of such cost convergence vectors, we can calculate the correlation between costs found after certain planner iterations. This is an estimate of how much costs after j_1 iterations are related to costs after j_2 iterations. On Figure 3.5 we try to illustrate visually the heuristic criteria we use for estimating the necessary amount of planner iterations to measure F from equation (3.12). We show correlations for 200 situations (in the grasping scenario that will be described later in detail in Section 3.7.3) for the first 100 iterations of the planner iLQG. It can be seen that after 60 iterations we get correlation of 0.95 with the final costs after 100 iterations, and this is a high enough correlation for our purpose. This means that we can use $j = 60$ iterations for measuring F for each combination of situation and initialization, because 60 iterations give a good estimate whether a certain initialization will lead the planner to good costs after convergence.

3.5.2 Learning the Situation to Motion Mapping

Once we have gathered data in the set D' as defined in Equation (3.13), we can use it to learn the trajectory prediction mapping from Equation (3.3). This is a supervised learning problem, but simply casting it as a multiple label classification problem is not the best way to approach it. It is too limiting to want to train using *exactly the single best* output label, i.e. trying to learn to map and transfer the best database movement $\boldsymbol{q}_{\hat{i}}$ for

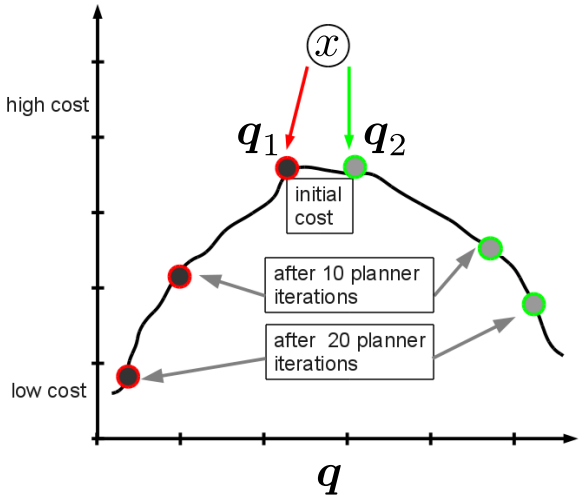


Figure 3.6: An illustration of costs vs trajectory values for $\boldsymbol{q} \in \mathbb{R}$. The planner takes an initial trajectory ($\boldsymbol{q}_1$ and $\boldsymbol{q}_2$) and gradually improves it over the initial cost $C(x, \boldsymbol{q})$. By using the convergence costs F (here after 2 iterations) for training the mapping f, we are more likely to predict $\boldsymbol{q}_1$, leading the planner to a lower cost optimum.

all situations x_j from set D' with the following loss counting[3] mis-classifications:

$$L^{\text{MULTICLASS}} = \sum_{x_j \in D_x} I(f(x_j) \neq \mathcal{T}_{x_j x_{\hat{i}}} \boldsymbol{q}_{\hat{i}}) \tag{3.15}$$

$$\hat{i} = \underset{(x_i, \boldsymbol{q}_i) \in D}{\operatorname{argmin}} \; F(x_j, \mathcal{T}_{x_j x_i} \boldsymbol{q}_i)$$

We would rather learn a predictive function that tries to predict a motion for a given situation likely to have low optimization convergence costs. It is not necessary to predict strictly the best output, but to have an output *label with low costs*. The image in Figure 3.6 illustrates the intuition behind trajectory prediction: we want to predict a trajectory initialization likely to lead to a low cost value *after* subsequent planner iterations.

In general the mapping from situation and initial motion to resulting cost after planning can be noisy. TP tries to predict the quality of planner initialization with some trajectory $\boldsymbol{q}_{\hat{i}}$. Until the final cost $C(x, \mathcal{O}_x \mathcal{T}_{x_{\hat{i}} x} \boldsymbol{q}_{\hat{i}})$ is calculated, we have the effect of the Transfer operator $\mathcal{T}$ and afterwards the effect of subsequent planner iterations $\mathcal{O}$, as shown in Equation (3.2). These two algorithmic modules, combined with the cost surface which is also nonlinear and with multiple local optima, make the learning problem quite complex. However, even with such noisy data one can get effective speed-ups of planning algorithms using TP, as we will show in Section 3.7.

The next subsections describe two possible approaches to learning such a mapping f leading to low cost motion planner initializations for the TP framework.

[3] Here $I(a \neq b)$ is the binary indicator function of a being not equal to b.

Nearest Neighbor Predictor (NN)

As in typical kernel machines, at the core of a good predictor is a good choice of similarity measure (kernel) in input space, see Schölkopf and Smola (2002). We consider rather basic prediction methods for the situation to trajectory mapping – namely nearest neighbor (NN) – but spend some effort in training a suitable similarity measure in situation descriptor space. We start with a descriptor vector x as input, which is potentially redundant and high dimensional, as explained in Section 3.3. We assume that similar situations have similar optimal trajectories. However, the usual notion of similarity as proportional to the negative Euclidean distance may not be the best for the high dimensional situation descriptors we have defined. We want to learn a similarity metric w in the situation descriptor feature space that selects appropriate features. In addition to improving the trajectory prediction quality, our learning method will allow to retain a compact set of the most representative descriptor dimensions by using a sparsity inducing L_1 norm.

We define the situation similarity function as:

$$k(x, x_i) = \exp\{-\frac{1}{2}(x - x_i)^T W (x - x_i)\} \tag{3.16}$$
$$W = \mathrm{diag}(w_1^2, .., w_s^2) \,,$$

$W \in \mathbb{R}^{s \times s}$ is a diagonal matrix, and s is the number of dimensions of the descriptor x. The nearest neighbor predictor f for x is

$$f : x \mapsto \mathcal{T}_{x x_{\hat{i}}} \boldsymbol{q}_{\hat{i}} \,, \quad \hat{i} = \underset{i \in D}{\mathrm{argmax}}\; k(x_i, x) \tag{3.17}$$

A direct formulation of a training loss for this predictor is

$$L^{NN}(w; D') = \sum_{i \in D} F(x, f(x)) \tag{3.18}$$

Because of the maximum operator in the predictor formulation in Equation (3.17) it is not possible to define directly a differentiable loss function for similarity metric learning from Equation (3.18). That is why we will define a probabilistic framework for the training loss: predict a trajectory with some probability proportional to the similarity to x. We define the probability to choose a specific trajectory $i \in D$ with such similarity function k as:

$$P(f(x) = \mathcal{T}_{x_i x} \boldsymbol{q}_i) = \frac{1}{Z} k(x, x_i) \tag{3.19}$$
$$Z = \sum_{i \in D} k(x, x_i) \tag{3.20}$$

Here Z is a normalizing constant. Afterwards we can calculate the expectation over the planner costs in situation x when initializing with Equation 3.19 as:

$$\mathrm{E}\left\{F(x, f(x))\right\} = \sum_{i \in D} P(f(x) = \mathcal{T}_{xx_i}\boldsymbol{q}_i) F(x, \mathcal{T}_{xx_i}\boldsymbol{q}_i) \tag{3.21}$$

Our goal is to find a similarity metric with low expected motion planning costs over all situations for which we have convergence information. We define the following loss function L using the cross-initialization data D':

$$L^{\mathrm{NNOpt}}(w; D') = \frac{1}{|D^x|} \sum_{x \in D^x} \mathrm{E}\left\{F(x, f(x))\right\} + \lambda |w|_1 \tag{3.22}$$

By minimizing this loss function we simultaneously improve the nearest neighbor predictor and do feature selection to select a small set of representative features. Our approach has some analogies with other kernel training and feature selection methods, e.g. Lowe (1995), where training parameters for radial basis neural network nodes also selects important features.

Having the squares of w on the diagonal of W in Equation (3.16) ensures that we get a positive-definite matrix W without requiring additional constraints for the optimization problem. The purpose of the L_1 regularization in term $\lambda|w|_1$ is to get sparse similarity metrics using only few situation features. We use the L_1 norm of w to get a sparse solution. We can take the vector of the ± 1 signs of w as the gradient of this norm, see Schmidt et al. (2007). This allows us use gradient based methods for minimizing the loss, and avoid using the more involved LASSO approach to sparsity (e.g. Tibshirani 1996). The exact calculation of the gradient of the loss is found in the appendix in Equation (A.1).

Simple extensions of the NN predictor are possible, namely taking not only the nearest neighbor situation and its corresponding motion, but taking the k-nearest neighbors. However, this would require modifications to our method in the NN predictor and in the Transfer IK operator itself, since we would somehow need to average the task space motions of a set of nearest neighbors. We tested several heuristics for this, the details of which we omit for brevity. Taking more neighbors $k > 1$ did not lead to performance improvement in practice, so we were satisfied with just the 1-nearest neighbor predictor NN as in Equation (3.17).

Trajectory Prediction via Cost Regression (SVR)

As an alternative to the above prediction scheme we also tested a cost regression approach (Tu and Lin 2010) to the cost -sensitive multiclass classification problem. For regression function we use the Support Vector Regression (SVR), a popular and powerful approach to function estimation, see Smola and Schölkopf (2004). We can learn regression models

$$f_i : x \mapsto F(x, \mathcal{T}_{xx_i}\boldsymbol{q}_i), \ , \quad x \in D^x, (x_i, \boldsymbol{q}_i) \in D \tag{3.23}$$

for the convergence costs F for each trajectory $\boldsymbol{q}_i \in D$ given some situation descriptor x using the cross-initialization data D'.

This method allows to use the SVR regression method with various kernels for the functions f_i to predict the convergence costs of initializing a planner with a trajectory from D in a given situation. Then we can use these multiple models to find the index of the trajectory with lowest costs. The SVR trajectory prediction model is defined as:

$$f : x \mapsto \mathcal{T}_{x x_{\hat{i}}} \boldsymbol{q}_{\hat{i}} \,, \quad \hat{i} = \underset{i \in D}{\operatorname{argmin}} \; f_i(x) \tag{3.24}$$

Here we call our whole setup of multiple regression models for classification the SVR predictor.

Comparison of the NN and SVR Methods for TP

We can calculate the computational complexity of the two algorithms in terms of the number of situations $d_x = |D^x|$ in the training set D^x, the number of trajectories $d = |D|$ in the motion set D, and the number of dimensions s of the descriptor vector x. The proofs of the following two propositions are found in the appendix.

Proposition 3.5.1 *The algorithmic complexity of minimizing loss 3.22 with a second order optimizer (BFGS) is $O(d_x ds + s^2)$.*

Proposition 3.5.2 *The algorithmic complexity for training the SVR predictor from Equation (3.24) is $O\left(d(d_x^\alpha + s d_x^2)\right)$ with $2 < \alpha < 3$.*

We estimate also the runtime complexity of predicting a trajectory for some new situation.

Proposition 3.5.3 *The algorithmic complexity of evaluating predictor 3.22 in one situation is $O(ds)$.*

Proof *We need to evaluate similarity to all d situation-trajectory pairs in set D, and each similarity measurement involves basic operations with a vector of size s. Thus the complexity is $O(ds)$.*

Proposition 3.5.4 *The algorithmic complexity of evaluating predictor 3.24 in one situation is $O(dd_x s)$.*

Proof *The parametric form of each such function after training SVR is $f_i(x) = k_x w_i$ where k_x denotes the vector containing kernel values $k(x, x_j)$ for each situation $x_j \in D^x$, and w_i is is the vector of coefficients found by the SVR. Evaluating each function is of Complexity $O(d_x s)$ and we have d such functions whose minimum we search, thus the complexity is $O(dd_x s)$.*

Note that s will be replaced in these formulas by some $s' < s$ if sparse feature selection is employed and only s' features are with weights different than 0.

An advantage of the SVR trajectory predictor from Equation (3.24) over NN is that it is a more powerful method, which can find well fitting models to the available data D'. The free parameters of the SVR predictor $\{\cup_{i \in D} w_i\} \in \mathbb{R}^{dd_x}$ are usually much more than the NN parameters $w \in \mathbb{R}^s$, because we expect to have more situations in the dataset than dimensions of situation descriptor x. This can lead SVR to more complex models, but also means that in some cases more data will be necessary to avoid overfitting (Russell and Norvig 2009).

A drawback of the SVR predictor from Equation (3.24) is that the set D defines a fixed set of cost functions f_i, whose minimum we predict. The predicted trajectory will always come from the set D, so we cannot generalize for motions outside of the set D. We also do not learn a general notion of situation similarity and cannot interpret the features meaningfully with this prediction method. To be more precise, we can get the notion of situation feature relevance for a *single* motion $\boldsymbol{q}_i \in D$, but this would have the meaning of situations likely to benefit from *exactly* this motion $\boldsymbol{q}_i$. In contrast, the feature relevance we get with the NN predictor is for *all* motions at once: a general notion of situation similarity and feature relevance.

Learning a similarity metric that describes well which situations have movements suitable for transfer has an interesting property: we transfer knowledge of expected costs for yet unseen movements, an action set of potentially unlimited size. Suppose we have a similarity metric w using data D' as defined in Equation (3.13). We have two interesting implications of this property which make TP with such a predictor flexible with respect to gathering train data and prediction in novel situations:

- we do not need to create the full cross-initialization dataset D': we can train the similarity metric also with missing entries $\big(x_j, x_i, F(x_j, \mathcal{T}_{x_j x_i} \boldsymbol{q}_i)\big)$.
- if we are given a novel situation x and some novel pairs of situations and optimal motions $(x_i, \boldsymbol{q}_i) \notin D'$, we can still reason about trajectories good for x by using the situation similarity metric w for the NN predictor.

3.6 Discussion: Trajectory Prediction and Imitation Learning

3.6.1 TP and Direct Policy Learning

The DPL approach to learning from demonstration (section 2.4.1) has some analogies with our trajectory prediction method, but also numerous differences. DPL tries to find a policy $\pi : s \mapsto a$ that maps state to action given observed state-action pairs (s, a). Given a parameterization of the policy, DPL is usually a supervised classification or regression problem. Usually the data comes from observation of an expert's (teacher's) behavior. No assumption of a cost function characterizing good motions is made, and the prediction is to be made only with the criteria to reproduce the expert motion accurately.[4] In

[4] See also our results in Section 5.5.3.

the terms of Call and Carpenter (2002) this would be *mimicry*: attempting to repeat an action without understanding its goal.

In the motion planning framework we can get large amounts of demonstration data from simulation, and use it to learn motion policies that can generalize to various situations. We do not need to reproduce the demonstrated movements perfectly with TP, since we assume there is a cost function as in Equation (2.3) specifying what good motions are, and a planner will refine these motions subsequently after the initial initialization by minimizing the costs. The essence of TP is to find trajectories that can lead such a planner quickly to good local optima of the cost function landscape. For this we need to predict a trajectory just once, in the starting situation, and rely that the structure in this trajectory will improve the planner performance. In the terms of Call and Carpenter (2002) this would be *goal emulation*: attempting to attain the goal of observed actions without caring whether the action is duplicated accurately. In the case of TP the goal is to obtain low cost motion planner output, which means usually going to a desired target configuration with low costs on the way. The predicted trajectory initialization is the action coming from our prediction policy, but it is transformed by additional planner iterations. The final planner output that the robot will follow in the world (the action of TP in another sense) can be quite different from the initial trajectory. This is acceptable, since only the low cost of the final trajectory matters for motion planning purposes. Penalizing deviation from the initialization is not a criteria in the cost function characterizing good motions for a task.

3.6.2 TP as a Macro Action Policy

Another important question is why we chose to have whole trajectories as prediction output. Predicting a whole trajectory means predicting a whole sequence of motor commands in joint space, a *macro-action* in the terms of McGovern and Sutton (1998). We can say that TP has a macropolicy to predict a macro-action, instead of the micropolicy used in DPL. A micropolicy would be in this case a mapping for every time step $\pi : x_t \rightarrow y_t$. Here y_t is the predicted movement command in some task space and x_t is the current situation descriptor, possibly changing at each time step. By iteratively predicting a movement y_t and recalculating the situation descriptor x_t after executing the movement, one can build whole trajectories.

The mapping of a situation to such a small local movement step is a challenging machine learning problem, since we have to account for global paths and the locally shortest path to the target is a dead end if the robot is trapped. The usual DPL approach will also learn a reactive mapping from state to control signal for a single time slice. This is less useful for global motion optimization, since such reactive policy can not anticipate and adapt for a longer time horizon.

A possible approach to remedy this would be to build networks of states connected via local actions (Stolle and Atkeson 2007). However, this can lead to jagged movements and fail to improve the planner behavior, as our results with RRT planners will show. Constructing such a network and searching for a global solution is also computationally

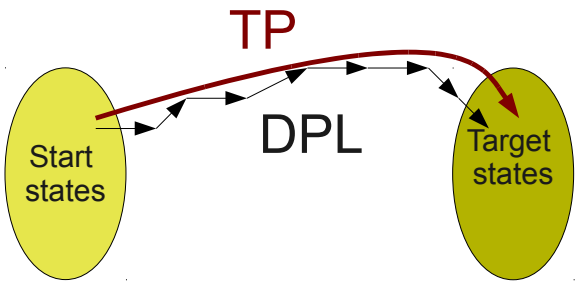


Figure 3.7: Trajectory Prediction as a macropolicy: predict a whole trajectory at once, in contrast to DPL which uses a micropolicy: predict 1 time step ahead (until the target is reached).

expensive. Another reason why micropolicies are more difficult to learn is that we would need to make accurate predictions for the whole state space, whereas for TP we need to learn mappings just for the smaller subset of starting situations in our dataset.

Using trajectories as macropolicies makes sense for our setup for several reasons:

- we use as input data the local motion planner output: whole trajectories $\boldsymbol{q}$ of length T steps and their costs
- we need to learn a predictive mapping saying which of these trajectories are good initialization for a given situation
- we need to output a whole trajectory $\boldsymbol{q}$ of length T to initialize a motion planner

Additionally, we assume that in situations where start and target robot configurations are a small subset of all possible configurations, a set of trajectories can be found that is a reasonable initialization for any start and target combination. This assumption holds if we select "reasonable" task spaces for TP, as indicated in section 3.4.1, and the experimental results in section 3.7 seem to confirm it.

Suppose we have a task where the starting configurations are a subset of all possible states, and the target configurations are another subset. TP will learn a (macropolicy) mapping from start configurations to whole trajectory pointing to the target configuration. The input domain of this mapping is a subset of all possible states. DPL, on the other side, will need to learn a mapping from all states to a motion command, because it uses a micropolicy that needs to predict 1 time step ahead. This micropolicy is called continuously until a target state is reached. This is illustrated in Figure (3.7).

3.7 Experiments

To test TP in practice we examined several simulated scenario setups for our robot model. Our robot is the Schunk LWA 3 arm with 7 Degrees of Freedom (DoF) and Schunk SDH hand with 7 DoF, making a joint configuration space $q \in \mathbb{R}^{14}$. The robot is shown in Figure 2.1(a), Figure 3.23(a) and Figure 4.7. For all scenario setups we examined, we generate random scenario instances (situations) by moving randomly objects around the

workspace, leaving the size and number of objects unchanged. Trajectory prediction learns from a set of demonstrated situations and movements and learns to generalize this behavior to new situations from the same generating distribution. For all tasks we planned kinematically with $T = 200$ time steps, which is a reasonable time resolution for movements lasting a few seconds.

All the cost functions were defined so that a movement with cost less than 0.5 is already quite good. The minimal possible cost is 0, and the costs are truncated to 2 if they exceed value of 2, since these are clearly bad movements and we do not want extremely high numbers to distort the results. As preprocessing for all prediction methods, we transform each dimension of the descriptors x to [0,1] by subtracting the minimum and rescaling, which improves predictive performance.

Dataset D had always 64 motions optimized until convergence for random situations. D' evaluated all these 64 motions for other 1000 random situations. For training the SVR predictor approach to TP we used a polynomial kernel of degree 4 and penalty parameter $c = 1$ with an implementation from the SHOGUN package Sonnenburg et al. (2006). For training the NN predictor approach to TP we used the Matlab optimization toolbox and the BFGS optimization algorithm for nonlinear optimization with gradient information and without constraints.

To validate the results for the predictor model f, for all experiments we split the set D' by dividing D^x in 800 situations for training and 200 for testing the predictors. This way we can reason about generalization ability of the predictive mapping in new unseen situations x. This can be also seen as transfer to new test situations of motions evaluated on the train set situations. For the creation of all these datasets we used the iLQG as local planner. For all planning algorithms and IK we used a C++ implementation on a Pentium 2.4ghz computer.

This section will proceed with a description of three different motion planning scenarios we examined. For each of them we will structure our work roughly in the following way:

- definition of the task costs and description of the structure of the situations and the objects present in the workspace
- description of the TP setup: what trajectory data was used, which predictor could learn the most accurate prediction model, and what structure was discovered in the data and situation features
- results for planning with TP with the best predictor: how local motion planners (AICO and iLQG) benefit and what speed-up in motion planning time can be obtained by using TP

3.7.1 Reaching on Different Table Sides

Scenario Setup

The first scenario we examined, contains the robot arm which has to reach a target across a table of size (1.2,0.7,0.1) meters with the finger as endeffector, around an obstacle (the

table) as seen in Figure 3.8. We controlled the 7 DoF of the arm, and the endeffector was defined as the tip of the hand. Different scenarios are generated by uniformly sampling the position of the table in a rectangular area of size (0.9, 02, 0.2) meters, the target in (0.5, 0.2, 0.6), and the initial endeffector position in (0.3, 0.3, 0.9). Situations with initial collisions were not allowed. Too easy situations where the endeffector was closer than 30cm to the target were discarded. This was done in order to avoid trivial situations and to put a greater focus on more challenging scenarios, where the endeffector must move on the other side of the table to reach the target.

We used the standard cost function in Equation (2.3) to generate motion fulfilling the following task requirements: reaching a target, penalizing collisions, keeping within joint limits and enforcing smoothness and precision at the endeffector position. We have chosen the term h to enforce a trajectory of short length with smooth transitions between the trajectory steps. We define h as

$$h(q_t, q_{t-1}) = \|q_t - q_{t-1}\|^2 \tag{3.25}$$

The cost term g in (2.3) is defined as

$$g(q_t) = g_{\text{collision}}(q_t) + g_{\text{reach}}(q_t) + g_{\text{limit}}(q_t) \tag{3.26}$$

where $g_{\text{collision}}$ penalizes collisions while executing the grasp movement. The value of this collision cost is the sum of the pairwise penetration depths c_i of colliding objects. Minimizing it moves the robot body parts away from obstacles.

$$g_{\text{collision}}(q_t) = 10^5 \sum_i c_i^2 \tag{3.27}$$

The task of reaching the target position with the endeffector is represented in g_{reach}. We want the target to be reached at the end of the movement, so we define this cost function to have a higher value for $t = T$:

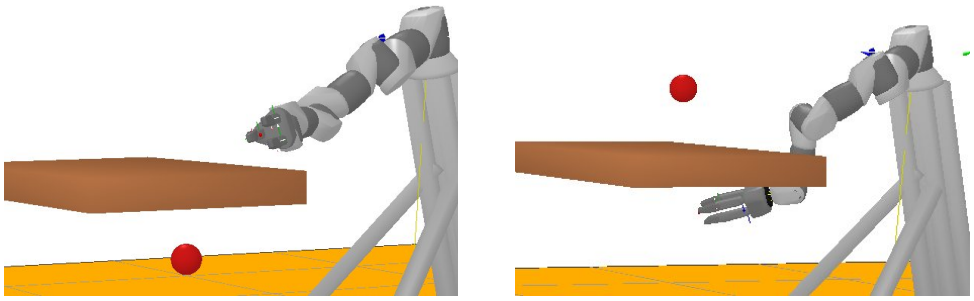
$$g_{\text{reach}}(q_t) = \begin{cases} 10^{-2} d^2 & t < T \\ 10^2 d^2 & t = T \end{cases} \tag{3.28}$$

where d is the Euclidean distance between the endeffector and the target.

The cost term g_{limit} puts limits on the joint angles:

$$g_{\text{limit}}(q_t) = 10^{-2} \sum_{i=1}^{n} \Theta(d_i - 0.1)^2 \tag{3.29}$$

where d_i is the distance of joint i from its limit (0 and 2 radians respectively), 0.1 is a margin, and Θ is the heavyside function.



(a) Situation x_a – reaching the target requires moving the hand under the table.

(b) Situation x_b – reaching the target requires moving the hand over the table.

Figure 3.8: Two situations; the goal is to reach the target with the robot hand.

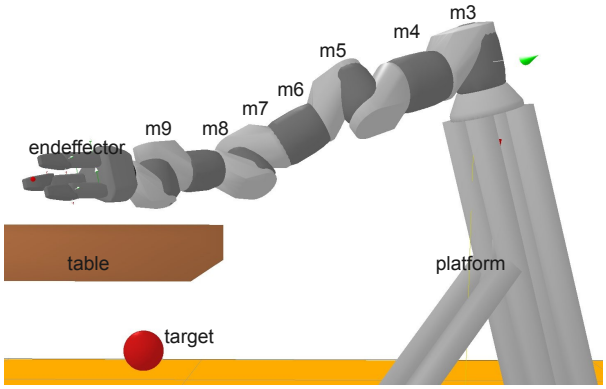


Figure 3.9: The 11 landmarks used for the descriptor x of the table reaching scenario: the centers of the 11 marked objects, 10 from the robot and a target object.

Trajectory Prediction Setup

Since we have only one obstacle in this table reaching scenario, we decided to use the geometric descriptor defined in Section 3.3.1 to predict trajectories using high-dimensional geometric information about few important landmarks. Concretely, the descriptor $x \in \mathbb{R}^{770}$ is defined as a 770-dimensional vector comprising a lot of the information relevant for this setup. We have defined 11 objects for which we measure pairwise geometric information: 7 segments of the robot arm, the endeffector, the robot immobile platform (similar to the world frame), the largest obstacle object (a single table in our scenario) and the reach target location, shown in Figure 3.9. 11 landmark objects give rise to 110 different pairwise distances, each of which is in $\mathbb{R}^7$.

The first demonstration set D had 64 random situations and optimal movements calculated for them using iLQG until convergence starting from a default initialization for reaching: a straight path from start endeffector position to target. We also examined the effect of smaller subsets from D, created using K-means clustering (MacQueen 1967) and Euclidean distance on the task space trajectories y. We used the medians of the

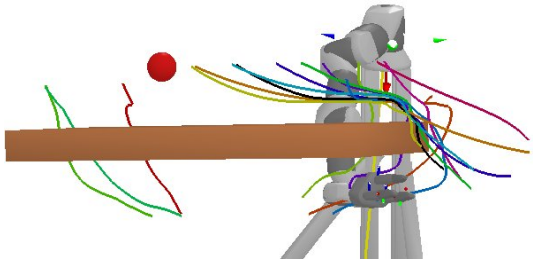


Figure 3.10: Visualization of endeffector movement trajectories from D in space Y_{obst}, centered on table.

clusters and situations for the creation of D' as in Equation (3.13) to provide initialization trajectories equal to the average task space trajectories for a cluster. By changing the number of clusters d we change the number of motions that can be output of TP. To learn the similarity metric and predictor f we measured the costs F of these initial movements $\boldsymbol{q}_i$ in all 1000 situations $x_j \in D^x = D$, using $j = 20$ iterations estimated by the heuristic described in Section 3.5.1.

We tested empirically 3 task spaces by creating different datasets D' for them. Figure 3.12(a) shows results for the usefulness for initialization of these spaces. The space Y_{target} represents movements relative to the target. The space Y_{obst} (with best performance in Figure 3.12(a)) consists of endeffector coordinates relative to the largest obstacle, see figure 3.10. This is a reasonable task space selection, since the main difficulty in this scenario lies in going around the edge of the table, which is always in a different translated position. Task space Y_{obst} provides a set of table avoidance paths, and predicting an appropriate table avoidance path for some situation is an effective motion initialization strategy. The joint space Q had poor performance, which confirms the hypothesis that joint space coordinates generalize poorly.

We will present results obtained by testing different prediction models on the cross-initialization set D'. Using this offline stored data, we can quickly cross-validate different models, without requiring expensive robot simulator calls. The set D^x is split in 800 situations for training and 200 situations for testing the cost of the best predicted trajectory.

Our choices for prediction methods, including both baseline and trained TP predictors, were

- NNOpt – trained NN predictor from Section 3.5.2, with $\lambda = 0.0001$
- NNEuclid – NN predictor without training, with $w = 1$ – default Euclid metric
- SVR – predictor via cost regression as in Section 3.5.2
- *best* – a predictor always taking the trajectory from set D^y with smallest cost F
- *mean* – a predictor choosing a random trajectory from set D^y

In Figure 3.12(b) we examine the performance of the different predictors f using data from the task space Y_{obst}. SVR and NNOpt have similar performance, and improve

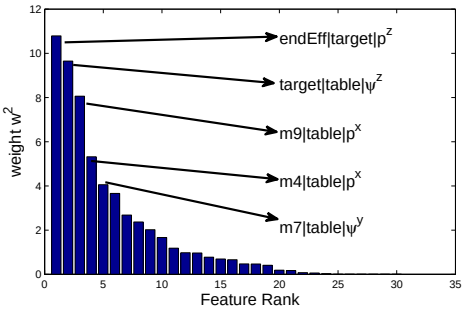


Figure 3.11: The 25 nonzero features in the learned metric NNOpt, and the geometric information of the top 5.

on both *mean* and NNEuclid. However, they are still away from the lower bound of performance *best*, which means that more complex models for similarity or regression can improve the performance further.

Both graphics 3.12(a) and 3.12(b) illustrate the trend that a bigger number d of trajectories in the set D can lead to better initializations. However, small numbers d provides already a variety of initial movements and allow good initialization with TP. Having few trajectories d also means having small cost for dataset D' creation. This can be tuned as necessary for different robot tasks with different computational costs of gathering data and complexity of the prediction task.

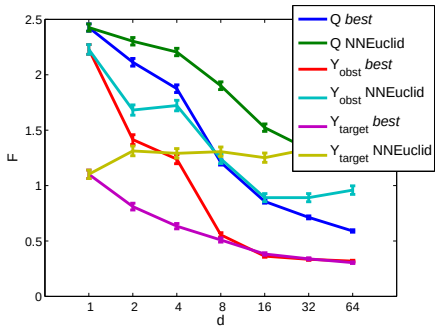
The regularization used for NNOpt also managed to compress the descriptor quite well: from 770 to 25 dimensions, as shown in Figure 3.11. The best features are the big table obstacle, the target, and the endeffector, which seems intuitively appealing interpretation of the reaching around table scenario.

We also tested varying the number of train situations d_x in the set D', as shown in Figure 3.12(c), and testing on the same 200 test situations. A difference between SVR and NNOpt is that NNOpt required significantly less training data for good performance. With as few as 25 situations on which all 64 movements are evaluated NNOpt can reach good prediction quality. In contrast, SVR needs at least 400 train situations to get good prediction quality on the validation set. This is to be expected, since the method SVR has many more free parameters than NNOpt, and needs more data for good prediction on unseen test data. This is an observation consistent with statistical learning theory and PAC bounds, see Russell and Norvig (2009).

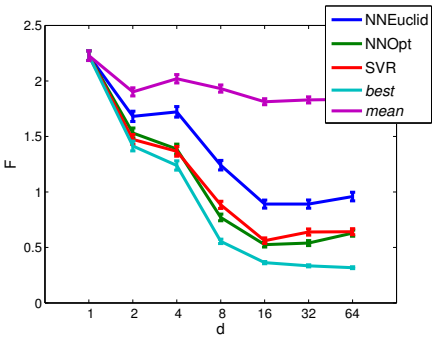
We fitted polynomials on the data shown in Figure 3.12(c) to see how the error decreases in terms of the used train situations d_x, see Raudys and Jain (1991) for details of this method of quantifying error in terms of train situations. For NNOpt the best error fit was in $O(\frac{1}{d_x})$, while for SVR the error decrease was slower - in $O(\frac{1}{d_x^{0.3}})$

Analysis of the Learned Representations with Kernel PCA

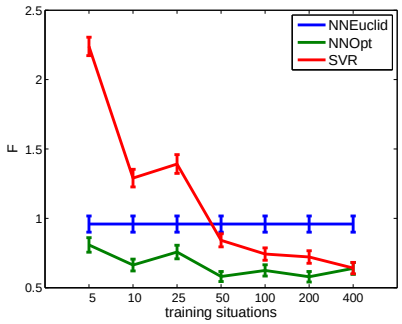
One way to illustrate the effect of learning the similarity metric is to use a low-dimensional embedding for visualization in the space of situation descriptors, see Figure 3.13. We use



(a) The 3 trajectory task spaces compared: joint space Q is worst.



(b) Different prediction strategies for $f(x)$ in space Y_{obst}: number of movements d vs convergence costs F.



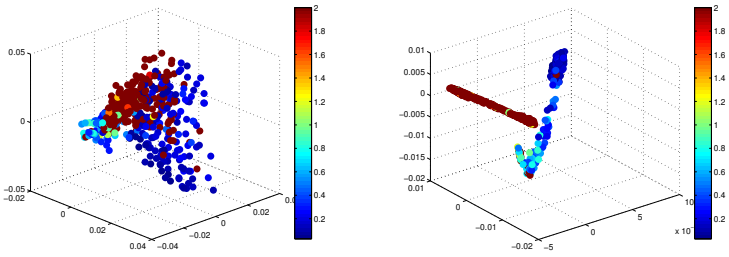
(c) Varying the number of situations used for training predictors $f(x)$ in space Y_{obst}.

Figure 3.12: Table reaching scenario: convergence costs F averaged over 200 test situations and using d motions for initialization.

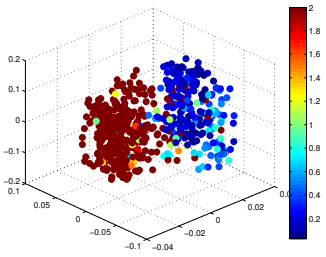
Kernel PCA (Schölkopf et al. 1998) on kernel matrices K calculated for 600 situations in the table reaching scenario, and each eigenvector u_i of K is one component of the new representation for all datapoints. By taking the 3 vectors with largest eigenvalues we get an effective low-dimensional visualization of the underlying high-dimensional situation space. We compare three different kernels:

- a Gaussian kernel $e^{-\frac{\|x_i - x_j\|^2}{\sigma^2}}$ using standard Euclidean metric, with $\sigma^2 = 0.1 \max_{i,j \in D}(\|x_i - x_j\|^2)$, corresponding to the NNEuclid predictor
- a Gaussian kernel $e^{-\frac{\|x_i - x_j\|^2_W}{\sigma^2}}$ using the learned NNOpt metric, with σ^2 chosen as above, corresponding to the NNOpt predictor
- the fourth degree Polynomial kernel $(x_i x_j + 1)^4$ used in the SVR predictor

The graphic in Figure 3.13 shows that the learned metric discovered structure in the space of trajectories. The two separated point groups in Figure 3.13(b) correspond to



(a) Gaussian kernel Euclid metric. (b) Gaussian kernel learned metric.



(c) Polynomial kernel.

Figure 3.13: Low dimensional embeddings of the situations using Kernel PCA. The color indicates the cost for initialization with one specific trajectory $\boldsymbol{q}'$.

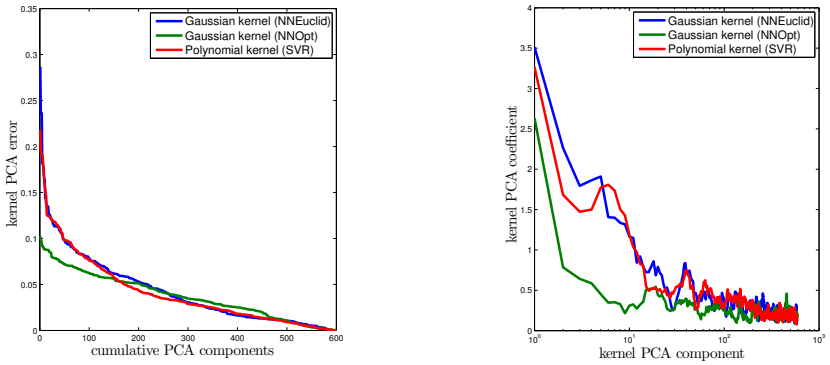
the two situation types where different initialization motions are appropriate, top-down or down-top (see Figure 3.8 for examples). The color of this visualization indicates the costs C after planner convergence for one motion $\boldsymbol{q}' \in D$ we chose. Again, it can be seen that the learned metric will likely improve NN prediction, since situations where $\boldsymbol{q}'$ lead to low planning costs were closer to another than with the Euclidean metric in Figure 3.13(a). The polynomial SVR kernel in Figure 3.13(c) is also able to discriminate well in which situation $\boldsymbol{q}'$ has low costs, but does not separate the two situation types (top-down or down-top) as well as NNOpt.

We also analyzed the quality of the above kernel matrices using the methods of Braun et al. (2008); Montavon et al. (2011).

- we denote Y as the column vector of costs of $\boldsymbol{q}'$ for all 600 situations, centered with mean 0.

- the kernel PCA coefficients for component i are defined as the absolute value of the dot product $|\, u_i^T Y \,|$. A large value indicates that the component contributes to label information, and usually having few large components and many close to 0 is an indication that noise is separated well from other components.

- the kernel PCA projection errors with d components are defined as the error of the



(a) Kernel PCA errors, adding cumulatively components.

(b) Kernel PCA coefficients for components.

Figure 3.14: Analysis of the situation kernel matrices. The axes are sorted by eigenvalues in descending order.

projected labels on $U_d = [u_i, .., u_d]$ w.r.t. the real labels:

$$\|Y - U_d U_d^T Y\|^2$$

An error curve quickly going to 0 usually indicates good kernel performance.

Figure 3.14 shows the result of this analysis:

- Figure 3.14(a) shows the kernel PCA projection errors. The NNOpt kernel has the lowest error of all methods when using the first 100 components. With more components the SVR polynomial kernel improves.
- Figure 3.14(b) of the kernel PCA coefficients also shows the good quality of the NNOpt kernel. Only the first few components have a large response to the label information – good signal-to-noise-ratio.
- Both Figure 3.14(a) and Figure 3.14(b) are consistent with Figure3.12(c): NNOpt makes good predictions even with very few samples, whereas SVR needs much more training samples to lower its error.

Planning Results

We present results for the average motion planning costs of the local optimizers and initializations as time progresses. From the point of view of a robotic scientist this would be the most important result: how does TP improve motion generation when evaluated on the robot platform. The results presented are for 200 random test situations on which we already validated the predictors in the previous subsection. We evaluated 6 different motion generation methods by combining different initializations and planners. We tested three different initialization methods:

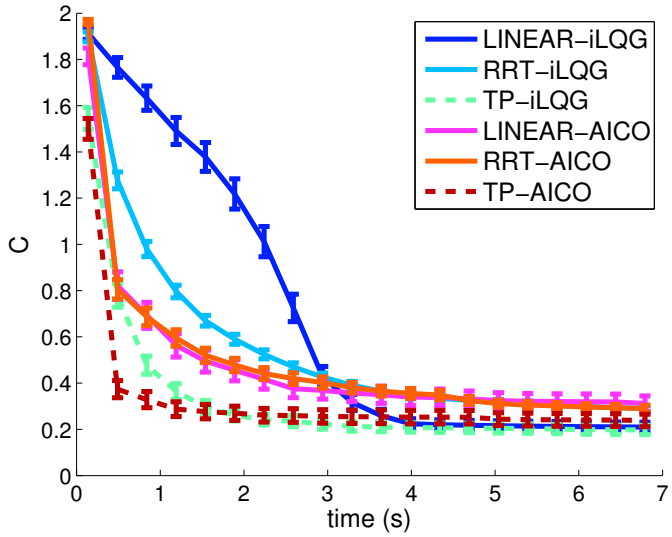


Figure 3.15: Performance of different methods in table reaching scenario: planning for 7 seconds. The average cost C of the planners during their convergence is plotted versus time in seconds.

- LINEAR
- TP
- RRT

LINEAR is the default option, where the start and goal endeffector positions are connected with a straight line path, which is followed by the robot hand using IK for initialization. TP uses the NNOpt nearest neighbor predictor in the task space Y_{obst}. Both trajectory prediction and straight line initialization require an IK operator from the endeffector path to joint space. The time for the IK operator ϕ^{-1} was 0.07s. The TP prediction itself is practically instantaneous. RRT initialization uses our implementation as described in Section 2.3.2. The RRT algorithm always works from scratch and does not require any training experience of the situations one can encounter. The creation of a RRT tree with 2000 nodes takes 8s, which is already a drawback for real time action and much slower than the other two initializations. However, we include RRT for performance comparison of the usefulness of such initial random collision free paths, ignoring this huge initialization time, assuming that some more efficient implementations of the RRT algorithm can do this faster.

The 3 initializations are combined with 2 different planner methods:

- iLQG
- AICO

For iLQG an initial trajectory $\tilde{\boldsymbol{q}}$ is already a part of the algorithm. For AICO we had to use $\tilde{\boldsymbol{q}}$ in a different way: only for the *first* iteration we used instead of the real task cost

Table 3.1: Summary of the timing of different operations for table reaching scenario.

operation	time (seconds)
create D (64 situations)	440
create D' (1000 situations)	89600
train NNOPt or SVR	40
1 planner iteration	0.07
TP prediction	0
TP and LINEAR Initialization	0.07
RRT initialization (4000 nodes)	8
TP-iLQG average cost 0.3	2
LINEAR-iLQG average cost 0.3	4

$C(x, \boldsymbol{q})$ a surrogate cost $\tilde{C}(x, \boldsymbol{q}, \tilde{\boldsymbol{q}}) = C(x, \boldsymbol{q}) + |\, \boldsymbol{q} - \tilde{\boldsymbol{q}} \,|^2$. This forces the solution to be near $\tilde{\boldsymbol{q}}$ and changes the belief states of AICO respectively. This is done just for 1 iteration, in order to add the initialization information to the Bayesian planner, afterwards planning continues in the normal way.

Both iLQG (Todorov and Li 2005) and AICO (Toussaint 2009) are local planners well suited for motion planning, as mentioned in the introduction. We set the iLQG convergence rate parameter $\epsilon = 0.8$; performance was robust with respect to different values of ϵ. For AICO we used instead of a fixed step parameter a second order Gauss Newton method to determine the step. One iteration of each of the planners took 0.07s, the bulk of which goes to collision detection and that is why the timings are similar for different planners.

We also tested direct gradient descent in joint space with the RPROP general optimization algorithm (Riedmiller and Braun 1993, Igel et al. 2005), but its performance was an order of magnitude worse than the other 2 planners, so we did not add it to the final results.

The results in Figure 3.15 show the convergence behavior of the planners for 7 seconds, and they allow us to make the following observations:

- TP is the best initialization for both AICO and iLQG, both speeding up convergence in the first initializations, and allowing to reach solutions with lower costs overall. Sometimes a first feasible solution is reached in less than a second for TP-iLQG, in comparison to 3 seconds for LINEAR-iLQG.

- TP-AICO also benefits greatly from a TP initialization. Note that the data D' for prediction was gathered only with iLQG planner data, so our predictors could transfer successfully to a new planner.

- The RRT path initializations are unexpectedly poor choices for planner initialization: they start collision-free and with lower costs, but they are difficult for the planners to improve and after some planner iterations even LINEAR finds better overall solutions.

- AICO is potentially very sensitive to initialization: with improper initialization (from any of the 3 initialization methods we examined) it can converge to bad

solutions, which are unlikely to be improved. iLQG is more robust in this sense: with more iterations bad solutions can still be improved.

In Table 3.1 we have summarized all timing involved in this experiments, including the average time needed to reach an average planning cost of 0.3 for all situations, corresponding to motions feasible on the average. This indicates the time savings one can achieve by implementing TP.

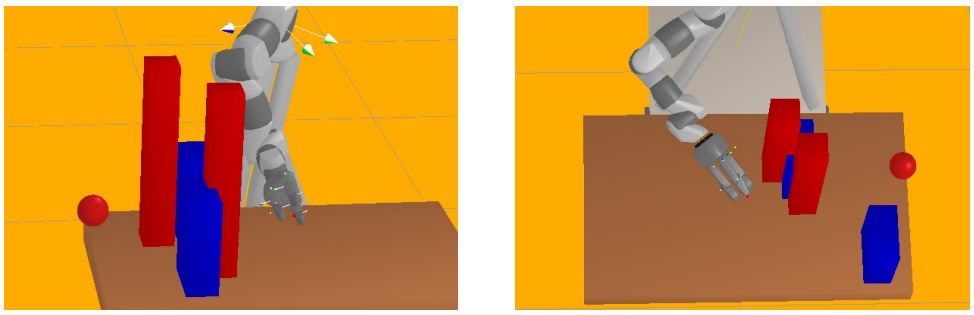
3.7.2 Reaching in Cluttered Scene.

Scenario Setup

In the next setup, the table (same object as in the previous experiment) was cluttered with 4 obstacles. Rectangles of various sizes and on random positions stand in the way of a target to be reached, as shown in Figure 3.16. We controlled the 7 DoF of the arm, and the endeffector was defined as the tip of the hand. The obstacle positions are randomly put over the table surface, and the target is put over the table to a place unoccupied by obstacles. We took the same reaching cost as the one defined in Section 3.7.1.

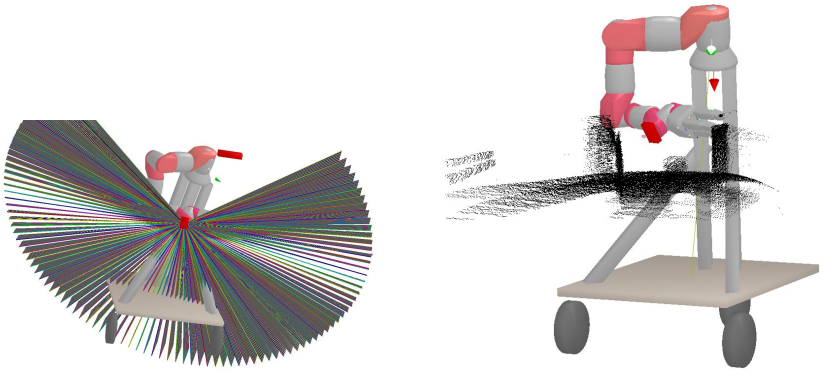
Trajectory Prediction Setup

For such cluttered situations we decided to test the sensor voxel descriptor $x \in \mathbb{R}^{413}$ from Section 3.3.2, since it is a compact way to represent the obstacle information. In the simulations we simulated an arm-mounted laser sensor delivering point cloud information to the scene, as in Jetchev and Toussaint (2010). We used a simple heuristic to gather information for the scene. A basic laser sensor gives information from a 2D plane. Before movement planning, the arm-mounted laser is rotated by moving the robot joint of the arm segment carrying it. A full rotation in 20 steps between the joint limits covers practically the whole workspace with rays. The points on objects intersected by rays are added as point cloud information to construct voxel grids, see Figure 3.17 for details.



(a) (b)

Figure 3.16: Typical situations on cluttered tables with red point as target, a table and 4 obstacles.



(a) Rotating the robot arm to cover space with rays.

(b) Convert the information for ray intersections with objects in a point cloud to be used as voxel descriptor.

Figure 3.17: Laser point cloud simulation used in the cluttered scenario.

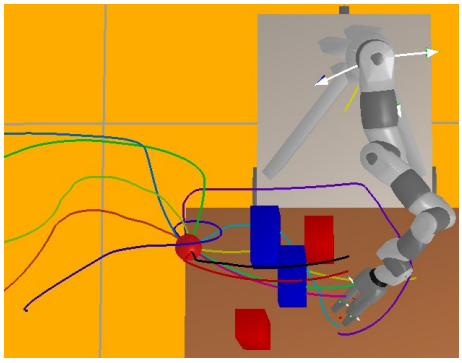
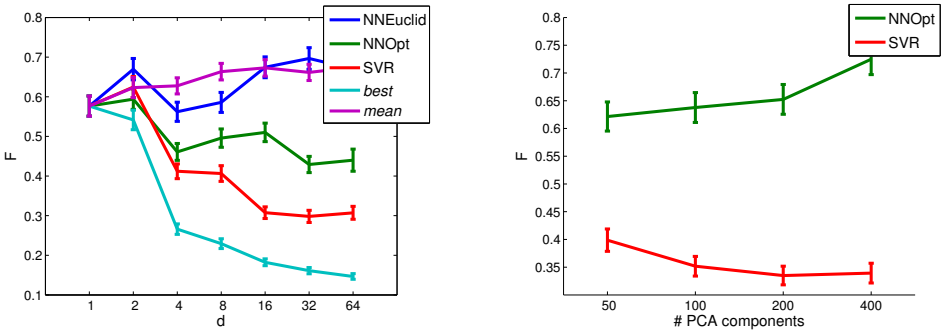


Figure 3.18: A visualization of different motion initializations in space Y_{target} (centered on target object) for a cluttered situation reaching task.

For task space we examined again the 3 choices from the previous experiment: joint space Q, table relative coordinates Y_{obst} and target relative coordinates Y_{target}, shown in Figure 3.18. Y_{target} worked the best for this scenario. Some of the situations required complex avoidance paths, so the linear initialization failed often to find any solutions. Thus for the database D of optimal movements we had to use RRT initialization, otherwise the dataset D' was created identically as in Section 3.7.1, again using $j = 20$ iterations for calculating convergence cost F.

In Figure 3.19(a) we compare the same 5 prediction methods defined in the previous section. NNOpt learned a much better predictor than NNEuclid. For this task the SVR approach worked significantly better than the nearest neighbor approaches NNOpt and NNEuclid. One possible explanation is that a standard distance metric is not appropriate for voxel descriptors, and probably other metric models or preprocessing of the volume data can help. Figure 3.19(a) also indicates the gains from using a higher number of motions d. 32 motions seemed to be enough to allow good prediction possibilities to TP,



(a) Different prediction strategies for $f(x)$: number of movements d vs convergence costs F.

(b) Varying the number of PCA components used for in the voxel descriptor vs convergence costs F (with $d = 64$ motions).

Figure 3.19: Prediction costs in the cluttered reaching scenario with task space Y_{target}.

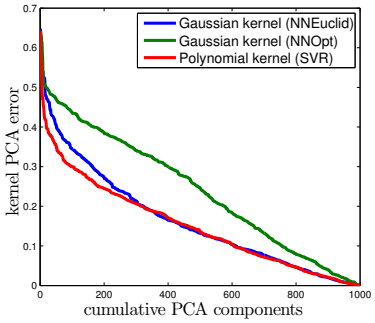


Figure 3.20: Kernel PCA errors for the cluttered reaching scenario; 3 different kernels are plotted. The polynomial kernel of degree 4 seems best.

doubling them up to 64 motions offered insignificant improvement.

In Figure 3.19(b) we examine how many PCA components are necessary to create voxel descriptors good enough for predictive purposes. With 200 components the SVR regression achieves the best result. The NNOpt method cannot handle well the voxel grid PCA components features and more components do not improve but worsen performance. We noticed also that simply looking at how well PCA compresses the input voxel data – 50 components have 96% of the variance, 100 have 99% and 200 components 99.8% of the variance – is not indicative of the TP performance with such a voxel descriptor.

In Figure 3.20 we show analysis of the three kernel matrices of NNEuclid, NNOPt and SVR evaluated on 1000 situations for cluttered reaching, with the same setup as in Section 3.7.1. The plot shows how the polynomial kernel has low label projection error with few components, and this is in agreement with our test results where the SVR predictor is better than both NN methods for the cluttered table scenario.

Planning Results

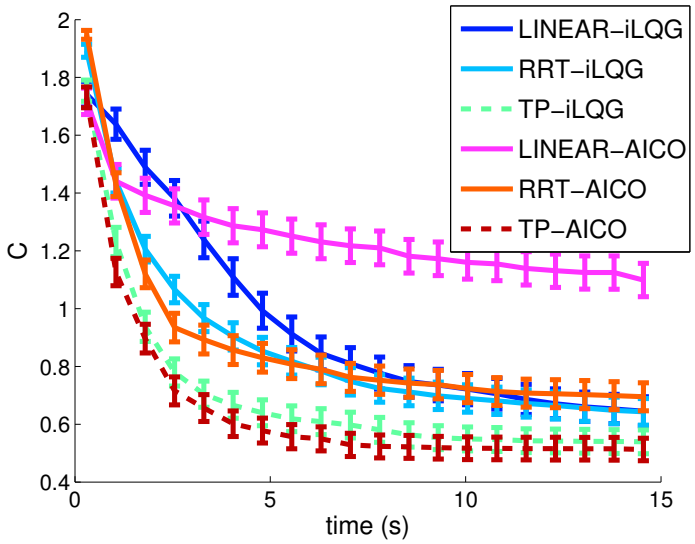
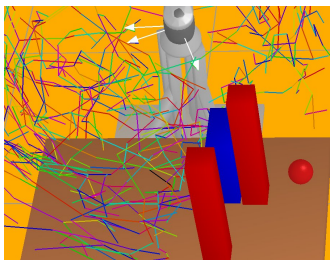
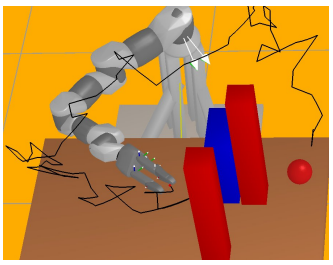


Figure 3.21: Cluttered scenario: the average cost C of the planners during their convergence is plotted versus time in seconds.

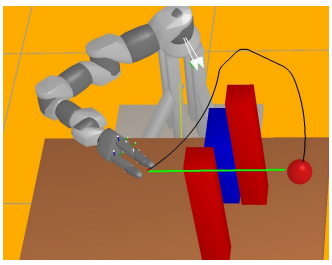
The results presented in this section are for 200 random test situations, different than the train situations. We tested the same 6 motion generation methods, but used instead of NNOpt the SVR approach for the trajectory prediction function in the task space Y_{target}. Each planner iteration and IK operation costs 0.15s. This is more than in the previous scenario due to more expensive collision check operations with more objects. This was the timing using the object models in the simulator. If we were to use the voxel representations obtained from analysis of point cloud data the timings would rise even more. Figure 3.21 summarizes our planning experiments, and Figure 3.22 illustrates how



(a) A RRT of random collision free samples accessible from the start position.



(b) The shortest path in this tree to the target is very inefficient if we want a smooth movement.



(c) A smooth movement from TP prediction (black). LINEAR (green) goes straight to the target and has high collision costs.

Figure 3.22: A visualization of different initializations for cluttered reaching scenario.

Table 3.2: Summary of the timing of different operations for cluttered reaching scenario.

operation	time (seconds)
create D (64 situations)	1472
create D' (1000 situations)	192000
train NNOPt or SVR	60
1 planner iteration	0.15
TP prediction	0
TP and LINEAR Initialization	0.15
RRT initialization (4000 nodes)	10
TP-AICO average cost 0.8	2
RRT-AICO average cost 0.8	14

exactly TP behaves differently than LINEAR and RRT initialization.

- For both AICO and iLQG planners, TP has lower costs than the RRT and LINEAR initializations. This is to be expected since LINEAR is often starting in collisions, see Figure 3.22(c).
- LINEAR-AICO is prone to failure even after many iterations: the many obstacles make for a highly nonlinear cost surface with multiple local optima, and AICO gets stuck in suboptimal solutions.
- Even though RRT takes significantly more time than the other initializations, it fails to find optimal motions for some situations optimally. RRT starts always with a collision free path, but suffers from the random nature of the path construction (Figure 3.22). RRT is worse than TP for the situations we examined.

We also tested a setup with 3 more obstacles, more difficult because obstacle avoidance paths become more complex. TP remained the fastest initialization even with this more cluttered setup, a transfer of useful behavior from the training setup with 4 blocks. This showed that the descriptors x and the predictor f can transfer knowledge to a more diverse set of scenarios without modification, since the occupancy of the workplace is represented well by the voxel descriptor x. On the other side, when considering the potential effect of adding even more objects in the scenario (e.g. more than 20), RRT has the best chance to solve such puzzles. The design of the scenario has big effect on performance.

In Table 3.2 we have summarized all times required for different operations involved in this experiment, including the average time needed to reach an average planning cost of 0.8 for all situations, corresponding to feasible if not yet optimal motions.

Hardware implementation

In addition to simulation, we also did hardware tests, as shown in Figure 3.23(a). We had robust performance in real scenes with different obstacles on tables (Jetchev and Toussaint 2010). We added to the Schunk robot setup the Hokuyo URG-04LX laser, see Figure 3.23(b), which is able to gather information (a scanner sweep) at 20hz. We first gathered

(a) The whole setup: robot, laser sensor and workspace cluttered with various obstacles.

(b) The Hokuyo URG-04LX laser used to gather point cloud information for a cluttered scene.

Figure 3.23: Hardware setup for the cluttered reaching scenario: the Schunk robot arm, the SDH hand and an arm-mounted laser.

data in simulation as in Section 3.7.2, and then used this data to predict fast initialization in real hardware situations. The real scene is not identical to the simulated scenes, but since it also has similar structure (a relatively large table and several objects on it in front of the robot) our scenario descriptors transfer adequately behavior. Such a coupling of simulated data and real applications can be quite effective for speeding up planning if the train and test situations come from a similar distribution, because creating huge simulated train sets is much cheaper than doing the same in hardware. A video of our robot reaching its target, as well as the datasets used for trajectory prediction, are available at `http://user.cs.tu-berlin.de/~jetchev/TrajectoryVoxel.html`.

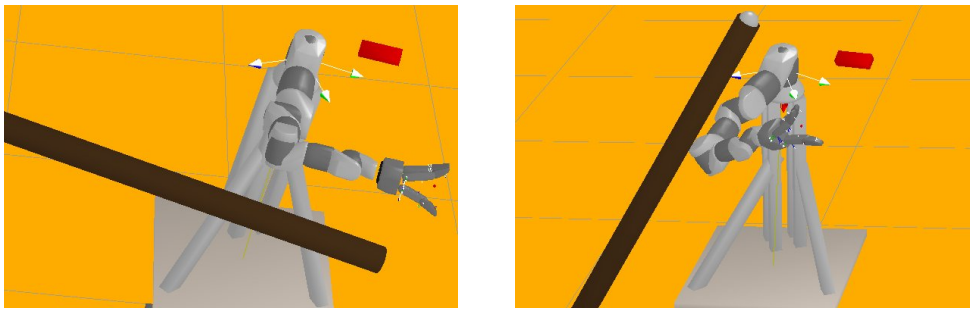
3.7.3 Grasping a Cylinder

Scenario Setup

In our last experiment we tested TP on tasks that are more complex than reaching. Grasping is one such task with a more complex cost function, and local planners have greater difficulty to solve this task. The grasp setup we devised contains a target cylinder 1.5m long and with radius 5cm that has to be grasped by the robot, see Figure 3.24. For the random situations we translated the cylinder center in a rectangular area $(0.9, 0.5, 0.4)$ and rotated it around its radial axis by random angles in $(0, 2\pi)$. We also moved the hand at random starting position similar to the previous scenarios. We controlled both the arm and hand for this setup, resulting in a joint space $q \in \mathbb{R}^{14}$.

The cost function we designed for grasping had the same smoothness term h, but a different term g:

$$g(q_t) = g_{\text{collision}}(q_t) + g_{\text{limit}}(q_t) + g_{\text{surface}}(q_t) \tag{3.30}$$



(a) (b)

Figure 3.24: Typical situations in the grasping scenario: the cylinder is rotated and translated randomly.

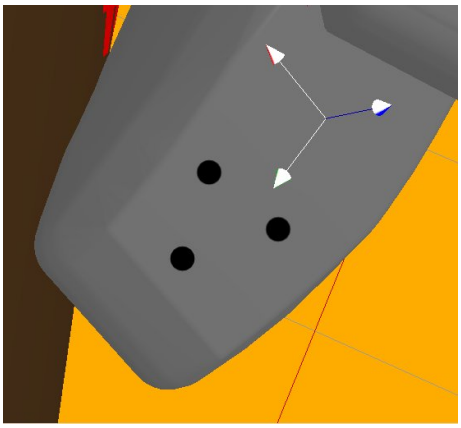
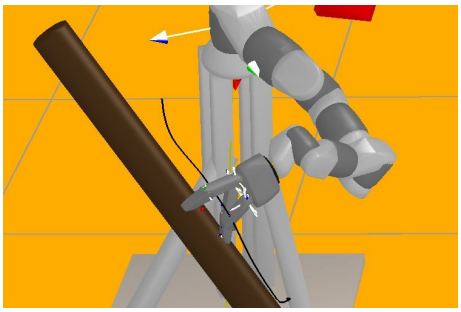


Figure 3.25: The 3 markers on each finger (indicated by black dots) used to force the fingers to move close to the object surface and align with it.

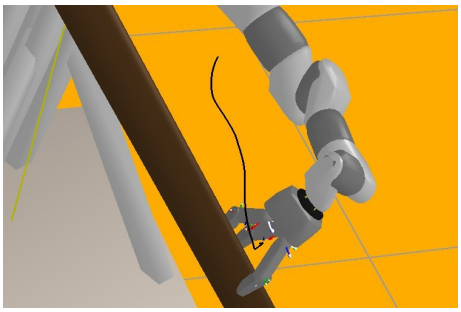
The collision and joint limit terms $g_{\text{collision}}$ and g_{limit} are the same, but the term g_{surface} is new. The term g_{surface} measures the distance from the target surface to some markers on the robot body and forces the robot to move these markers on top of the surface of the target object. We defined 12 such markers, 3 on each of the 3 robot fingers, and 3 on the wrist, as shown in Figure 3.25. By taking a configuration of 3 markers near the surface of the fingers we force the robot to also align the surface of the fingers with the grasp target object surface: the distance is minimal to all markers only if the surface of the fingers is parallel to the object surface, which leads to better grasps. By using the 3-marker configuration we can approximate this alignment property just by using proximity task variables and not requiring more complex vector orientation task variables.

The definition of g_{surface} is:

$$g_{\text{surface}}(q_t) = \begin{cases} 10^{-3} \sum_{i=1}^{18} \eta_i^2 & t < T \\ 10^{2} \sum_{i=1}^{12} \eta_i^2 & t = T \end{cases} \tag{3.31}$$



(a) A grasping movement: first approach the object.



(b) Finally the fingers should close on the object surface

Figure 3.26: Grasping scenario: a good grasping movement with two phases. The black line visualized the robot hand trajectory.

where each η_i stays for distance to target cylinder surface of each of the 12 markers. This cost function is similar to the one used by Dragiev et al. (2011), and it reflects the simple assumption that the robot holds the object tightly if the fingers all grasp it. The condition that the fingers also align with the object surface is useful, since the geometry of the robot hand makes this possible only if the fingers oppose one another, an even stronger heuristic criteria for a good grasp. The target can be grasped anywhere with this cost, but the challenge lies in positioning the robot fingers on the surface without colliding with it. Indirectly this requires two movement phases: first approach object and open hand, then grasp. Figure 3.26 shows these two phases.

Trajectory Prediction Setup

We used here the geometric descriptor as in Section 3.7.1, but with a slightly different object set: the 7 robot arm segments, the endeffector, the target cylinder center targetC, and a marker on top of the cylinder targetE. This results in 90 pairwise object distance descriptors and a situation descriptor $x \in \mathbb{R}^{630}$.

The sizes of datasets D and D' were as in the previous experiments, with the only difference being that we needed $j = 60$ planner iterations (using the heuristic of Section 3.5.1) to measure the convergence cost F, which made data gathering slower.

We examined two task spaces. First, the joint space $Q \in \mathbb{R}^{14}$. Second, $Y_{\text{qhand+target}} \in \mathbb{R}^{10}$ consisting of the 7 hand joints and the 3D relative position of the arm in the target frame. $Y_{\text{qhand+target}}$ is a better task space than q, as illustrated by Figure 3.29(a). The relative positions of the hand in the target frame generalize well to target rotations and move the hand to positions which can be grasps near the cylinder surface, and the finger joint information moves the fingers in an appropriate pregrasp shape.

In Figure 3.29(b) we compare the 5 different predictors for good trajectory. The SVR predictor was the best, closely followed by NNOpt.

The regularization used for NNOpt also managed to compress the descriptor quite well: from 630 to 17 dimensions, as shown in Figure 3.27.

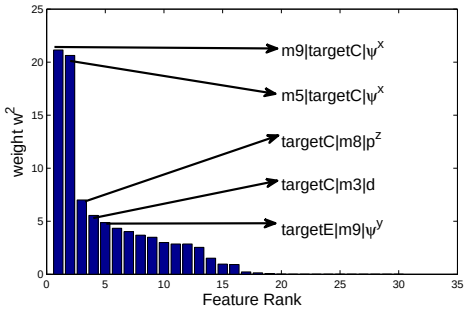


Figure 3.27: Grasping task: the 17 nonzero features in the learned metric NNOpt, and the geometric information of the top 5 features.

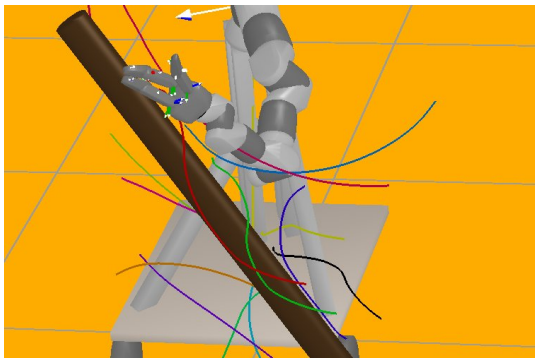


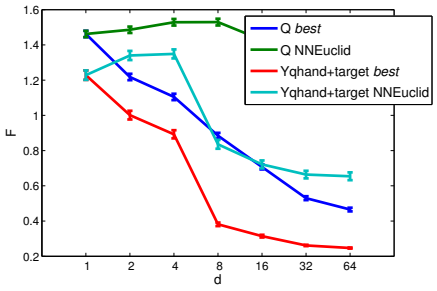
Figure 3.28: Grasping scenario: a set of trajectory initializations in space $Y_{\text{qhand+target}}$, consisting of arm and hand movements relative to the target frame. Only the arm movement is visualized.

Planning Results

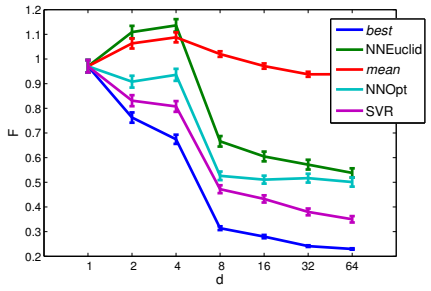
We tested 4 combinations of planner and initialization: AICO and iLQG combined with LINEAR initialization (with target the center of the cylinder) and TP initialization using SVR for prediction. We did not test RRT for grasping, since it would require major modifications to the default RRT algorithm.

In this scenario the time for a single planner iteration and IK transfer was 0.15s, and optimization to convergence required sometimes as much as 100 iterations and was with a high failure rate, so this problem has the potential to gain a lot from TP. Figure 3.30 shows our results for this more complex task:

- the combination TP-AICO finds the best solutions overall: 2 seconds planning time with the TP initialization vs 14 seconds for LINEAR-AICO for reaching cost 0.4.
- TP-iLQG is similarly superior to the default LINEAR-iLQG: 8 seconds when using TP-iLQG, while LINEAR-iLQG never reaches such average costs.
- for the grasping task AICO is better than iLQG, regardless of the initialization used.



(a) The two trajectory task spaces Q and $Y_{\text{qhand+target}}$: number of movements d vs convergence costs F.



(b) Different prediction strategies for $f(x)$: number of movements d vs convergence costs F.

Figure 3.29: Costs of different methods in cylinder grasping scenario in the task space $Y_{\text{qhand+target}}$.

A possible explanation of the last point can be the specific character of the grasping cost we used. For grasping the challenge is to coordinate multiple body parts to do a more complex movement, whereas for the previous two tasks the challenge is to avoid collisions. It seems that the inference algorithm of AICO can handle complex motions better than collision avoidance. It is also worth noting that the TP predictor was trained using data created *only* with the iLQG planner. Both D and the cross-initialization dataset D' were evaluated only using iLQG, and thus the predictive mapping was trained to speed-up iLQG. However, as our results indicate, AICO benefits also greatly from using the structure inherent in a good situation appropriate initial trajectory. We can call this property *transfer* on the level of different planners.

Table 3.3 summarizes the time costs for different operation in the grasping scenario. Despite the huge cost for gathering data needed for training TP, the gains when using TP for initialization make it worth it when faced with a new grasping situation. As we mentioned in Section 3.5.1, usually for more complex tasks with costly planner iterations data gathering for TP is more expensive, but also the possible speed-up from good initialization is much greater.

Table 3.3: Summary of the timing of different operations for the grasping scenario.

operation	time (seconds)
create D (64 situations)	1920
create D' (1000 situations)	576000
train NNOPt or SVR	40
1 planner iteration	0.15
TP prediction	0
TP and LINEAR Initialization	0.15
RRT initialization	-
TP-AICO average cost 0.4	2
LINEAR-AICO average cost 0.4	14

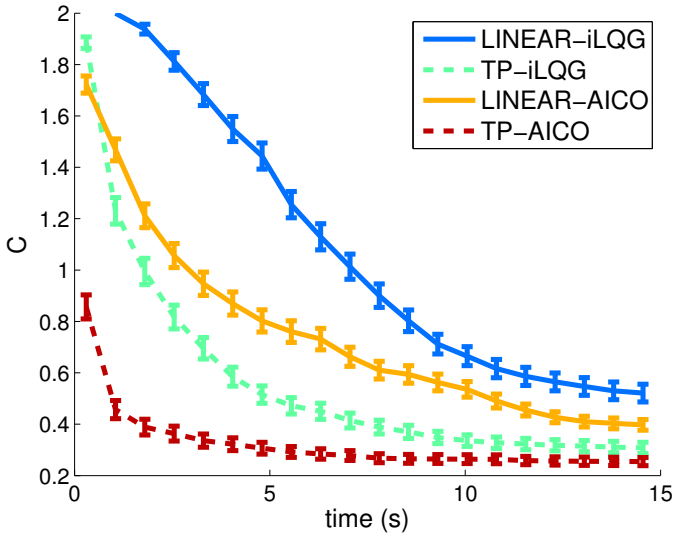


Figure 3.30: Performance of different methods in cylinder grasping scenario. The average cost C versus time in seconds.

3.8 Conclusions from TP

In this thesis chapter we proposed a novel algorithm to improve local motion planning methods - Trajectory Prediction (TP). TP can exploit data from previous trajectory optimizations to predict reasonable trajectories in new situations. We proposed three key aspects to solve this problem: an appropriate situation descriptor, task space transfer of previously optimized trajectories to new situations, and cost-sensitive classification. Concerning the situation descriptor, we demonstrated that learning a (L_1-regularized) metric in a high-dimensional descriptor space significantly increases performance of the mapping. Interestingly, this means that we can extract features of a situation (e.g., choose from a multitude of possible coordinate systems) that generalize well w.r.t. trajectory prediction. The extracted features allow for an intuitive explanation of the crucial latent factors to choose one movement over another, and also allow to visualize the structure of situation space. The task space transfer – that is, first projecting an old trajectory to a task space and then projecting it back in the new situation – allows an adaptation to the new situation implicit in the inverse kinematics.

Speeding up local planners is crucial for fluid robot interaction with the world and humans. The TP framework for movement prediction is of great practical utility for many motion planning tasks, as shown by our experiments. A good initialization makes the planner converge faster. Additionally, the planners can converge to potentially better solutions which are not likely to be discovered by a naive initialization not using the experience of movement and situations incorporated by the trained predictors.

The results in three different setups, a simple scene with one obstacle, a complex cluttered scene modeled by a 3D sensor, and an object grasping task, show clearly the

utility of trajectory prediction for speeding-up the motion planning. The most complex task we examined, object grasping, gained the most from the experience of previous motions learned from TP, with a speed-up of almost 7 times faster planning times on average.

TP is a flexible method that makes transfer on at least two different levels: transfer from train to test situations, and from training with one planner (iLQG) to testing with another planner (AICO). This flexibility and transfer of good trajectory predictions can be of great utility in situations when motion data from different sources is available.

3.8.1 Future Work

Future work will focus on making trajectory prediction less dependent on designer choices. In our current implementations, selecting and testing a pool of reasonable task spaces for motion transfer is important for the performance of the method. Developing data-driven methods for finding such task spaces using the demonstrated optimal motions will be a step further toward understanding the latent structure of motions. Chapter 5 of this thesis will deal with methods that can potentially make this step in the future. It is also possible to change the representation of trajectories from a sequence of (joint) states in time to a more compact parametrized representation such as splines (Zhang and Knoll 1995).

Another possible direction is applying trajectory prediction to more complex and realistic scenarios, with sensor uncertainty and moving obstacles in the workspace under strict time constraints. Speeding up motion planning in such situations will be of even greater utility, especially if combined with parallel exploration of alternative predicted trajectories. A framework for such trajectory prediction in parallel will be presented in the next Chapter 4.

Chapter 4

An Extension of Trajectory Prediction: Parallel Process Planning

The motion generation by planning framework, described in Section 2.3 and utilized in Chapter 3, works efficiently if we accept the general limitation that some time (a few seconds) is needed to calculate a plan before any motion starts. We call this time the *planning time*. If we assume that the world will not change during the planning time when the robot is passively calculating a plan between the start of planning and the start of motion execution, this delay is not a problem. However, if we have a world where obstacles can change suddenly on a time scale smaller than the duration of the planning time, the usual planning approaches will be of great disadvantage. A reactive forward controller does not have the disadvantage of this passive initial behavior, but it creates motions of inferior quality usually that can fail to achieve some tasks. In this chapter we will present a parallel online planning framework that improves upon these drawbacks of local planning and is a practical solution to planning in dynamically changing scenarios. We call this the Trajectory Prediction Parallel Framework (TPPF). It has three components:

- an online planning algorithm building upon local planning algorithms
- an algorithm to maintain multiple local online planners in parallel and select the best one to control the robot
- a predictive initialization module to speed-up planning convergence of each of the local planner threads

The use of online planning gives the advantage of greatly reduced window between planning and acting. Our approach maintains multiple movement trajectories optimized in parallel CPU threads and can improve upon and switch between alternative trajectories online *during* the motion execution by the robot. Such use of parallel processing hardware is a further advantage of our approach, speeding up planning even more. As trajectory planners find only locally optimal solutions (see Chapter 3), the third advantage of our parallel framework is the exploration of different planning process initial-

izations. The diversity of such initializations turns out to be crucial for overall planning performance. We use trajectory prediction (TP) to generate varying initializations of good quality. Empirical evaluations demonstrate the benefits of our approach TPPF in different motion problems with dynamic changes in the positions of obstacles and targets.

4.1 Introduction

Consider the problem of robotic motion planning in a dynamic environment, where obstacles or task constraints change over time in an unpredictable manner, e.g. a robot arm in a factory where other robots and humans move around constantly. In such cases, the usual two steps of planning a motion trajectory and then executing it will not lead to efficient behavior. We want the system to react to such changes of the situation by adapting online the plan. In some cases, a change of situation implies switching to an alternative motion which was previously suboptimal but now became better than the current motion plan. For instance, going on the left side of an obstacle can be good at first, but if the object changes speed and starts moving to the left, switching to a motion that circumvents it to the right is required immediately, a nonlocal plan change. In such scenarios the motion generation system is required (i) to do online motion planning and (ii) to explore and optimize multiple (locally optimal) trajectory alternatives in parallel. In this chapter we present a framework called TPPF for such parallel online trajectory optimization. It can be viewed as a natural extension of trajectory prediction utilizing parallel processing and efficient heuristic for mixing planning and acting. More specifically, there are two core motivations for such a system:

- Fast trajectory optimization methods are typically local and may require multiple restarts with different initializations to produce high quality solutions. This motivates a parallel online trajectory optimization system even in static environments.

- In dynamic environments it is advantageous to maintain multiple locally optimal motion hypotheses in parallel. Even when the robot controller only follows the currently best plan, an unexpected switch in situation (change of the target or obstacles) may render a previously suboptimal hypothesis to become the best. When we maintain such multiple alternatives the robot may quickly switch to a better hypothesis – leading to a very fast reaction to situation changes.

It turns out that the crucial issue for parallel online optimization is the *initialization* of the different parallel optimization processes. Clearly, this initialization must ensure the diversity of the alternative trajectories, as well as allow for fast optimization. The latter implies that the initializations should be close to local optima, that is, we need a heuristic to generate ad-hoc trajectories likely to be good depending on the situation. To ensure diversity, the heuristic should offer not only one guess but a variety of qualitatively different trajectories. Our approach is to use trajectory prediction (Jetchev and Toussaint 2009) to propose such initializations in a given situation. A second interesting issue in

the context of parallel optimization is, how can all optimization processes be kept *consistent* with the real motion of the robot. It does not make sense for an optimization process to maintain a trajectory which is not consistent with the current real position of the robot, i.e. does not have this position as the starting point. Since the robot moves online following only one specific trajectory, all alternative trajectories have to be modified on the fly to stay consistent. Our parallel planning framework TPPF solves these issues and is an efficient way for motion planning in time-sensitive tasks.

This chapter proceeds with an overview of related motion planning methods in Section 4.2. Then we introduce our motion optimization framework in Section 4.3. In Section 4.4 we describe the trajectory optimization component and how it fits into TPPF. In Section 4.5 we describe our test simulation setup and the motion planning problems we want to solve. We also present our test results, both in simulated random environments and with real hardware. Our conclusions are in Section 4.6.

4.2 Related work

A basic module of our framework are local trajectory planners, see Section 2.3.1. Typically these methods precompute optimal trajectories or regulators before the movement is executed, requiring some planning time. Our framework will employ them with a modification that allows to plan while executing the first time steps of trajectories, effectively resulting in online planning. We build on very fast local optimization methods, but exploit trajectory prediction and parallel optimization of multiple alternatives to find better solutions using the limited available time.

The sampling based planners (Section 2.3.2) also suffer from long planning times, and are at disadvantage for motions requiring fast reaction time and adaptation. The use of parallel computing to speed up randomized planning has been investigated in Caselli and Reggiani (1999); Challou et al. (1995). These works split the load of building the sample tree among CPU cores, and speed up the average time to get a solution for motion tasks. However, such a speed up of randomized search still does not allow simultaneous moving and planning with dynamically optimal trajectories (our approach). Second, in order to be able to react quickly and avoid rebuilding the whole sample tree from scratch in dynamic situations, additional procedures are required to reuse such networks of obstacle free nodes when the world environment changes Ferguson et al. (2006), but the computational time remains significant.

An initial collision free plan can be adapted to moving obstacles by changing the plan to keep it in the free space by combining global path planning with "elastic bands" that can be shaped and stretched (Quinlan and Khatib 1993; Brock and Khatib 2000). Such adaptation of the current motion hypothesis is comparable to *one* of our optimization processes. However, it does not address the parallel adaptation scenario, the diverse initialization issue and the consistency maintenance of alternative trajectories.

4.3 Planning and Parallel Trajectory Exploration Framework

4.3.1 Notation

We define the robot configuration as $q_t \in \mathbb{R}^N$, the joint posture vector. Let $\boldsymbol{q}_{0:T} = (q_0, .., q_T)$ be a movement trajectory starting in time step 0 and with time horizon T. In the previous Chapter for TP we wrote for notational simplicity just $\boldsymbol{q}$ without he time indexes of the trajectory start and end time steps. However, the parallel framework we are describing now will require to specifically work with these time indexes, so we introduce them as new notation. In a given situation x, i.e., for a given initial posture q_0 and the positions of obstacles and targets in this problem instance, the motion generation problem is to compute a trajectory which fulfills different constraints, e.g. an energy efficient movement not colliding with obstacles. We will define a the concrete choice of x when we describe our experiments in Section 4.5.3. We formulate the planning task as an optimization problem by defining a cost function $C(x; \boldsymbol{q}_{0:T})$ that characterizes the quality of the joint trajectory in the given situation and task constraints. A local planner like iLQG will try to find the best movement (or its regulator) for a given situation.

To arrive at the optimal trajectory $\boldsymbol{q}^*_{0:T}$ many local optimizers start from an initial trajectory $\boldsymbol{q}_{0:T}$ (usually the output of some heuristic) and then improve it, e.g. by making steps in direction of the cost function gradient. The planners iterate such steps until convergence.

4.3.2 Framework and Algorithm

Computers have evolved recently to have more and more CPU cores, and this presents opportunities to speed up motion planning. Multiple processes in different processor cores can calculate movement plans simultaneously, effectively multiplying processing power and allowing both faster motion generation and lower trajectory costs. This gives the opportunity to devise algorithms that use these processes to efficiently distribute the computations required for motion planning and control. We design TPPF to have one control process and k optimization processes running in parallel, as illustrated in Figure 4.1.

The control process stores the currently best planned trajectory $\boldsymbol{q}^*_{0:T}$ and controls the robot joints in real time. At the same time the k processes optimize k different trajectories $\boldsymbol{q}^k_{0:T}$ and evaluate their costs $C(x; \boldsymbol{q}^k_{0:T})$.

Trajectory prediction gives k different initial trajectories, which are then optimized in parallel. This initialization module is called at time $t = 0$ and we can reinitialize at a later time point when new alternative trajectories are required. Trajectory prediction for this parallel setup will be examined in Section 4.4.

Let us describe the algorithmic procedure of the online parallel optimization in detail. We assume that the optimizers output a reference trajectory $q_{t:T}$ – later we will briefly explain the case when the optimizers output a sequence of Linear Quadratic Regulators. We start at time $t = 0$. The control process has no input from optimizers at all and controls the robot to stand still, that is to follow the constant trajectory $q^*_{0:T} = q_0$. Each

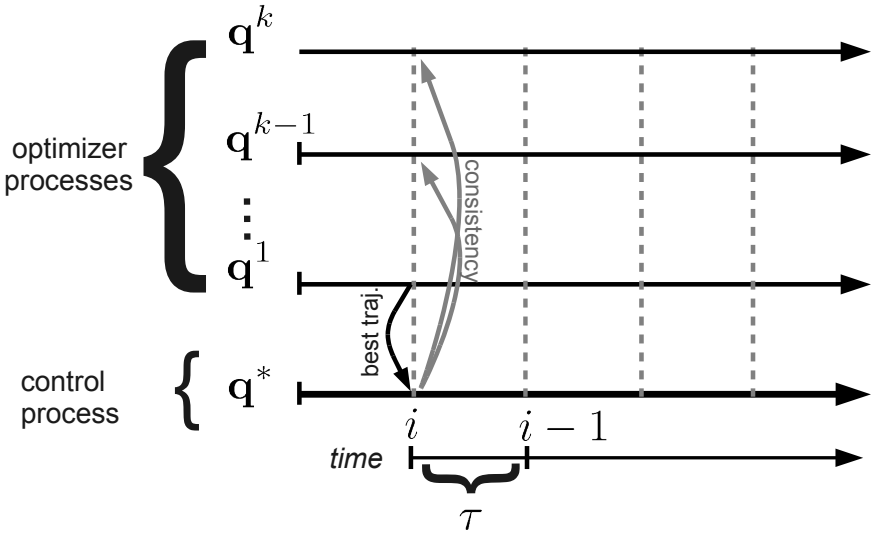


Figure 4.1: Time view of the parallel framework: dashed lines mark the intervals of duration τ, at which two things occur: (i) the control process selects the best optimized trajectory (ii) all $\boldsymbol{q}^k$ are modified to become consistent with the expected $\boldsymbol{q}^*_{i\tau:T}$. Between the dashed vertical lines the control process follows $\boldsymbol{q}^*_{(i-1)\tau,i\tau}$ and the optimizers improve $\boldsymbol{q}^k_{i\tau:T}$. In the illustrated case, the first process $k = 1$ was best at time $t = \tau$.

of the k optimization processes is initialized with a predicted trajectory $\boldsymbol{q}^k_{0:T}$ (see Section 4.4). We assume that one optimization step takes real time τ. Hence, when the optimizers provide updated trajectories the real time will be $t = \tau$ and the expected state of the robot will still be $q^*_\tau = q_0$. Therefore the optimizers are queried to optimize trajectories $q^k_{\tau:T}$ which start at $q^k_\tau = q^*_\tau$ and are of duration $T - \tau$.

After one iteration of the planners the real time is $t = \tau$. We expect all optimizers to have finished one optimization step and output their trajectories $q^k_{\tau:T}$. The control process does the *minimal cost selection*, that is, the best of these trajectories is assigned to the control process: $q^*_{\tau:T} \leftarrow q^k_{\tau:T}$ with $k = \underset{k}{\operatorname{argmin}}\ C(x, \boldsymbol{q}^k_{\tau:T})$. The control process will follow this trajectory during the real time interval $[\tau, 2\tau]$ and we expect the robot to be in state $q^*_{2\tau}$ at time $t = 2\tau$. We need to update all optimization processes to account for this: The currently maintained trajectories $q^k_{2\tau:T}$ of the k optimizers are in general inconsistent with the expected state $q^*_{2\tau}$, that is $q^k_{2\tau} \neq q^*_{2\tau}$. Each optimization process does a *consistency modification* of its current trajectory according to:

$$\forall k:\ \forall_{t\in 2\tau:T}:\ q^k_t \leftarrow q^k_t + \frac{T-t}{T-2\tau}(q^*_{2\tau} - q^k_{2\tau})\ . \tag{4.1}$$

This can be viewed as simply making each alternative trajectory start at $q^*_{2\tau}$ to be valid. Figure 4.2 illustrates this procedure in a simple two dimensional case. Once all different trajectories are consistent, the optimizers continue with another iteration of improving the remaining trajectory $\boldsymbol{q}^k_{2\tau:T}$, subject to the constraint $\boldsymbol{q}^k_{2\tau} = \boldsymbol{q}^*_{2\tau}$.

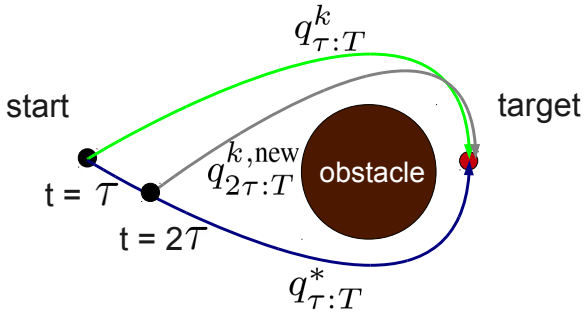


Figure 4.2: A sample situation where maintaining two alternative motions improves planning. At time τ the control process follows $\boldsymbol{q}^*_{\tau:T}$ (blue) around the obstacle. The kth optimization process maintains the alternative trajectory $\boldsymbol{q}^k_{\tau:T}$ (green). At time 2τ the robot's expected position has changed and $\boldsymbol{q}^k_{2\tau:T}$ is modified to $\boldsymbol{q}^{k,\text{new}}_{2\tau:T}$ (grey) to become consistent with $\boldsymbol{q}^*_{2\tau}$. In the case the object might move towards $\boldsymbol{q}^k$, it will be advantageous to have maintained this alternative.

Generally, in the ith planner iteration the time is $t = (i-1)\tau$. The control process does a minimal cost selection of all provided trajectories – this gives a current reference trajectory $\boldsymbol{q}^*_{(i-1)\tau,i\tau}$ for the time interval $[(i-1)\tau, i\tau]$. The expected robot state at real time $i\tau$ is $q^*_{i\tau}$; each optimizer does a consistency modification of its current trajectories using Equation (4.1); the optimizers improve the trajectories $\boldsymbol{q}^k_{i\tau:T}$ and will be finished before real time $i\tau$. The pseudo code of this procedure is given in Algorithm 2. For simplicity we examined the case when we use iLQG as a local planner, which can directly use an input trajectory in joint space to start planning. We can also consider a more complicated setup using a motion regulator instead of such trajectory, see Section 4.3.2.

Note that all calculations of costs of planners use the world state, and TP initialization requires a descriptor x of this world state. In TPPF we assume we get constant information from some sensors of the world, which allows to update the world state and descriptor x, and change planner and TP initialization behavior.

The LQR planning case

Let us discuss the case when the optimizers output a time series of Linear Quadratic Regulators instead of a reference trajectory, e.g. iLQG Todorov and Li (2005). A LQR at time t is determined by a vector $q_t \in \mathbb{R}^n$ and a quadratic form $Q_t \in \mathbb{R}^{n\times n}$ such that the control signal u_t (e.g., the step in joint angle space in the kinematic control case) can be expressed as

$$u_t = Q_t(q_t - q^{\text{real}}_{t-1}) \,. \tag{4.2}$$

where q^{real}_{t-1} denotes the true state of the robot at time step $t-1$. The q_t can be interpreted as a reference and Q_t as a gain matrix. The optimizers now output a reference trajectory $q_{t:T}$ and a series $Q_{t:T}$ of gain matrices. The online parallel optimization scheme described above is unchanged except for (i) the control process now simply implements this LQR, and (ii) the consistency modification can now be done more naturally since the

Algorithm 2 Online Parallel Trajectory Optimization

1: calculate the situation descriptor x
2: initialize all $\boldsymbol{q}^k_{0:T}$ using trajectory prediction
3: **for** $s = 0 : T - \tau$ **do**
4: wait until real time $t = s\tau$
5: calculate the situation descriptor x
6: select $k^* = \underset{k}{\text{argmin}}\, C(x; q^k_{s\tau:T})$ and reassign the control reference $q^*_{s\tau:T} \leftarrow q^{k^*}_{s\tau:T}$
7: **for all** $k = 1 : K$ **do**
8: optionally, reinitialize $\boldsymbol{q}^k_{(s+1)\tau:T}$ using trajectory prediction
9: modify $\boldsymbol{q}^k_{(s+1)\tau:T}$ to become consistent with the expected state $q^*_{(s+1)\tau}$
10: optimize $\boldsymbol{q}^k_{(s+1)\tau:T}$ for one planner iteration, subject to $q^k_{(s+1)\tau} = q^*_{(s+1)\tau}$
11: **end for**
12: **for all** $k = 0$ **do**
13: follow the control reference $\boldsymbol{q}^*_{s\tau:(s+1)\tau}$,
14: runs parallel to the optimization threads
15: **end for**
16: **end for**

optimization output is a regulator instead of a reference trajectory. In the case of DDP or iLQG we only have to set the constraint $\boldsymbol{q}^k_{i\tau} = q^*_{i\tau}$ before forward iterating the regulator and thereby computing the consistent reference trajectory $\hat{\boldsymbol{q}}_{i\tau:T}$ at which the local approximations are computed before the backward Ricatti backups.

4.4 Adapting Trajectory Prediction to the Parallel Framework

The straightforward approach to initialize the optimization processes from Section 4.3.2 would be to take a linear start-to-goal endeffector path and translate it to joint space using Inverse Kinematics (IK). However, such linear initialization can often lead to bad solutions, see Section 3.7. Another drawback is that the standard procedure does not create a diversity of alternative trajectories to overcome the locality of the optimization and explore alternatives in parallel. We propose to use trajectory prediction as a heuristic to generate a diversity of initialization trajectories that explores different typical motion patterns.

We will use experience in the form of demonstrated trajectories in different situations to get better planner initializations and thus a better movement policy. The implementation of TP requires a dataset $D = \{(x_i, \boldsymbol{q}_i)_{i=1}^d\}$ comprising pairs of randomly generated situations x (drawn from a distribution of reasonable world situations) and trajectories $\boldsymbol{q}$ optimized offline with a local planner. The situation descriptor x is modeled as a vector containing information for the world situations, see Section 4.5.3 for the concrete implementation for TPPF. Once we have such data, we can learn a function $f : x \mapsto \boldsymbol{q}$ that predicts the costs of initializing a planner in situation x with some motion transferred from the trajectory database. In the SVR trajectory prediction implementation (Section

3.5.2) the predictor had the form $f(x) = \mathcal{T}_{xx_{\hat{i}}}\boldsymbol{q}_{\hat{i}}$, $\hat{i} = \underset{i \in D}{\text{argmin}}\ f_i(x)$, where we first learned to predict expected costs when initializing a planner with motion $\boldsymbol{q}_i$ as $f_i(x)$. This prediction method can be used to provide the most likely trajectories for a given situation as mentioned in Section 4.3.2: output $\boldsymbol{q}_k$ with lowest predicted costs $f_k(x)$.

In relation to the parallel framework described in the previous section, TP has two advantages:

- It is more likely to speed up each planner with a good initialization. The prediction method and transfer operator require time roughly equal to a single planner iteration. Considering the benefit of TP to make planning faster by an amount of time equal to multiple iterations (see Section 3.7), TP can be a very effective method in conjunction with the parallel framework TPPF.
- With multiple differently initialized processes TP will explore a wider diversity of possible trajectory initializations and increase the chance that in any given situation there will be some proposed movements that can be later refined to low cost solutions. This adds reactive adaptation capabilities to TPPF so that if the world situation changes abruptly, one of the different trajectories being optimized in planners will be close to a good solution.

We expect that increasing the number of planning processes from 1 to 2 will have great benefit for the quality of motion plans. However, after a certain number we expect that the performance gains from such multiple processes will flatten and there will not be any significant increase in performance. This is consistent with our results in the previous chapter and Figure 3.12(a) which shows how the number of motion alternatives in a set influences the quality of the best of them. For example, the performance gain of having 2 instead of 1 trajectories is larger than the gain of having 64 instead of 32 trajectories.

4.5 Experiments

4.5.1 Robot and Planner Setup

We examined a reaching task for our robot setup (Schunk LWA3 arm and a SDH hand) shown in Figure 4.7, both in simulation and in hardware. We set in our experiments a trajectory time resolution $T = 200$, a time horizon of 200 slices of 0.01 seconds each, for a total duration of 2s. We controlled only the 7 joints of the robot arm. The robot hand endeffector has to reach a point target location at the final time step of the trajectory, i.e. at time $t = T$. For all experiments we used the canonical reaching cost function as defined in our previous experiments in Section 3.7.1. In general, costs less than 0.1 are excellent motions, costs around 0.4 are feasible without collisions but probably near constraint limits, and costs above 0.6 are with collisions or fail to reach the target.

For our parallel framework and for training trajectory prediction we always used iLQG as motion planning optimizer module with $\tau = 0.07$ seconds (time per iteration)

on a 2.4 GHz CPU. We expect other planners to perform also reasonably well in such a framework. The inverse kinematics routine required by both the default linear initialization and the IK Transfer $\mathcal{T}$ takes also τ seconds. The main cost bottleneck for both iLQG and IK are the collision checks, required for cost calculation at each of the T time slices.

4.5.2 Setup of Two Different Scenarios for Reaching Task

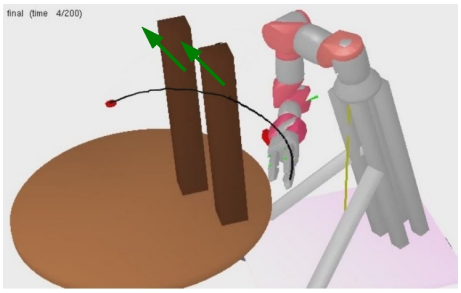
For the same reaching task we designed two scenario domains, for each of which we generated random world scenes x with specific obstacle placement structures. We chose these scenario domains to highlight two different aspects of humanoid robot motion planning. The first is online planning with dynamic obstacles. The second is online planning in situations where the targets randomly change position – the sequential target domain. For the simulations of both scenarios we assumed perfect noise-free 3D sensors that provide information for the speeds and positions of obstacles and objects. Given the initial position and velocity of an obstacle and assuming there are no external forces to accelerate the obstacle, our simulator can predict its position. However, if something changes like the velocity of the object, which causes the estimates to be changed. Such a change of estimated object positions will most likely change the costs of a trajectory $\boldsymbol{q}$. More formally, $C(x; \boldsymbol{q}) \neq C(x'; \boldsymbol{q})$ if x represents the simulator state before and x' the simulator state after the change. Such points of change are likely to benefit from quick change of the planners' currently maintained trajectories.

Dynamic Obstacle Scenario

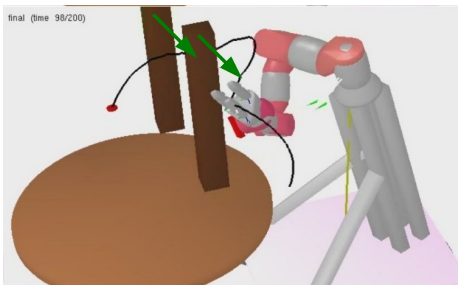
The robot has to reach a target (red point) with its hand and avoid two obstacles placed between the initial hand position and the target. The obstacles start with some velocity vector and change it to a different velocity after random t_{change} seconds in the time interval [0.5, 1.5]. We limit the movement range of the obstacles between the robot and the target, neither hitting the robot, nor flying out of reach, to have challenging and still feasible planning problems. The initialization module is called at times 0 and $t_{\text{change}} + \tau$, where new paths are initialized, in response to the obstacle behavior change (assumed to be detected from sensors updating the situation descriptor x).

We chose to have only one velocity change for the obstacles in this domain. Generalization to multiple change points is straightforward, by just calling the TP initialization module in the parallel architecture at the times of change.

Figure 4.3 illustrates an example scenario in this domain. At first, the obstacles move away from the robot, and the initial plan is a rather straight approach to the target. At time t_{change} the obstacles change speed on a collision course with the initial plan. Quick re-initialization allows the robot to immediately retract slightly and find a better path to the target out of the obstacles' way.



(a) Planned motion q^* in the early phase of the trial: collision free *if* the obstacles would continue their motion.

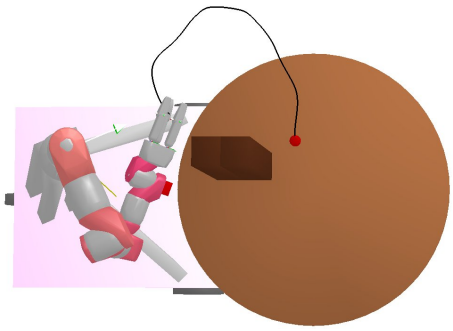


(b) At t_{change} the obstacles change velocities. The controller selects an alternative trajectory that retracts the hand and then circumvents the obstacles.

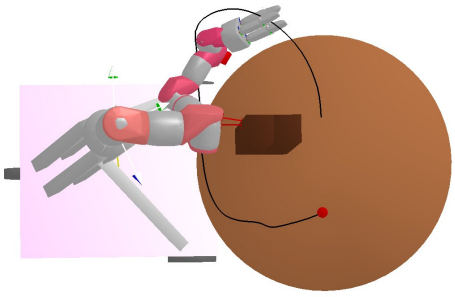
Figure 4.3: Typical situations in dynamic obstacle scenario: reach target and avoid moving obstacles.

Sequential Target Scenario

In this scenario the robot has to reach a point target around one static obstacle. The scenarios are generated in two steps. First, the robot starts from its initial position with the hand withdrawn and has 2 seconds to reach the randomly placed target. Then, the target is moved at a random position *on the other side of the obstacle* and the robot has again 2 seconds to reach it. The target is deliberately placed to confuse the robot. The difficulty is that in some situations even if the target appears very close to the robot hand, the direct move is blocked and the robot has to take a long detour to circumvent the obstacle and reach the target on the other side, see Figure 4.4.



(a) Initial position - reach target.



(b) Second phase - reach new target location.

Figure 4.4: Typical situations in sequential target scenario.

4.5.3 Trajectory Prediction Setup

We used the SVR predictor variation of TP (see Section 3.5.2). For each problem domain, we generated 1000 random situations and calculated trajectories optimized until convergence with iLQG. We then "compressed" this dataset by clustering the trajectories in 20 clusters (see Section 3.7.1 for details on this procedure). We used the joint space Q for task space for TP in the sequential target domain. For the dynamic obstacle domain we used the task space Γ, which is basically a 3D space of the hand endeffector coordinates in a specially constructed frame, see Jetchev and Toussaint (2010) for details. This frame is constructed with basic geometric operations (scaling, translation and rotation) to always have the start hand position at $(0, 0, 0)$ coordinates and the target at $(0, 0, 1)$. This design of task space has the following interesting property: the backprojected paths by Transfer IK (Section 3.4.2) would *always* start at the current hand position and end at the target position. This is the most important cost term for the reaching task, and satisfying it by design of task space turned out to be advantageous in our case, see Figure 4.5

To define situation descriptors x we designed a descriptor having the features we assumed were relevant for the scenarios we try to solve. We define x in the following way

$$x = (z_1, z_2, z_3, z_4, z_o, z_o^T, z_{\text{target}}) \in \mathbb{R}^{21} \, . \tag{4.3}$$

z_1, z_2, z_3, z_4 are the current 3D positions of the robot's hand, wrist, elbow and arm. z_o is the current obstacle position and z_o^T is the estimated obstacle position at the final time T. z_{target} is the target location. Thus, the descriptor for the sequential reaching scenario with one obstacle the descriptor x has 21 dimensions. For situations with two obstacles we simply added additional coordinates for the other obstacle: z_{o_2} for current state and $z_{o_2}^T$ for expected position at the final time step T. For the dynamic obstacle scenario with 2 obstacles the descriptor x has 27 dimensions.

We use a simple piecewise constant velocity motion model for the obstacles of the dynamic obstacle scenario. More complicated motion models with acceleration can easily be incorporated by modifying the descriptors x and adding additional features. There is related work on visual cue processing for motion prediction in cognitive science Coull et al. (2008), where the features of motion necessary to make accurate predictions for object movements and potential collisions are discussed, e.g. polynomials of speed, acceleration and position values. In more complex situations cluttered with numerous obstacles, a voxel situation descriptor Jetchev and Toussaint (2010) will be a better choice.

4.5.4 Results

Results for Dynamic Obstacle Scenario in Simulation

We compared the performance of 4 different methods. The baseline is iLQG with linear path initialization LINEAR. We tested also linear initialization with a hypothetical iLQG that makes 2 iterations in 0.07 seconds, and called this LINEAR2. The assumption behind this test is to illustrate the potential effect of a faster local planner with only one default

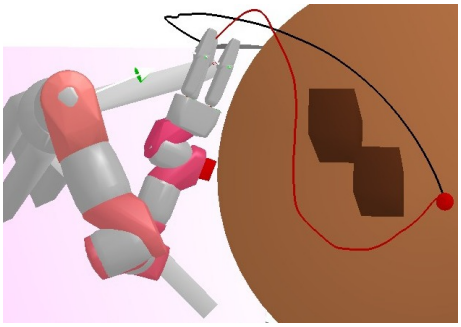


Figure 4.5: Two hand movements transferred from task space Γ: each start at the current position and end at the red point target.

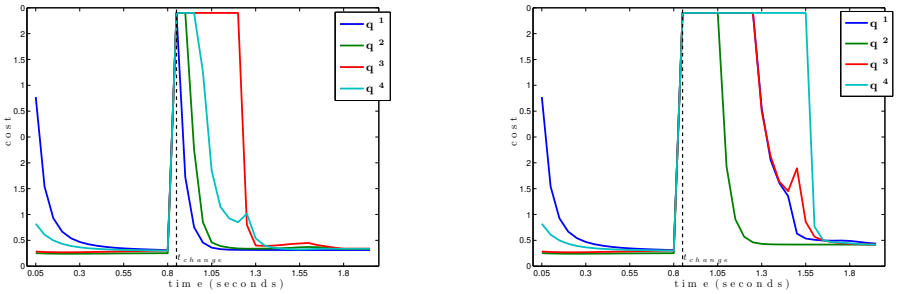
initialization. Trajectory prediction and parallel planning was labeled TPn, denoting n parallel processes, in addition to the the control process.

For quantitative results we examined costs in a set of randomly simulated scenarios with different obstacle and target placements. We will give the average costs over multiple runs and standard deviations for the mean estimation. Keep in mind that all costs show the quality of the actual movement done in the fixed time of 2 seconds for planning and motion execution. Thus the times of all planner and initialization combinations are the same, and what can really distinguish them is the quality of motions generated.

In the first column of Table 4.1 we examined the effect of having a re-initialization at the velocity change time t_{change}:

- All TPn can solve such motion problems effectively, while the standard approach of LINEAR has too high costs.
- TP1 has lower costs than LINEAR2 and this shows that initialization with an appropriate initial path is more effective than using a linearly initialized but 2 times faster planner.
- Using parallel exploration with more processes (TP2, TP3, TP4) can further lower costs. However, the gains of TP4 over TP3 are quite small, and we expect that adding more parallel processes (e.g. TP5) will have diminishing effect on performance improvement.

Not reinitializing with trajectory prediction at t_{change} lowered performance for all methods, as the second column of Table 4.1 shows. If the world situation x changes suddenly as in our dynamic obstacle domain at time t_{change}, all of the process costs jump abruptly. Local improvements from planner iterations cannot fix the different old plans fast enough in the short period of 2 seconds for adaptation. Reinitializing is advantageous at such change points, because trajectory prediction can predict diverse movements again and increase the chance that some of the planned movements can lead to good optimal trajectories. Without new initialization the old trajectories can not be "repaired" quickly enough and collisions can occur driving the costs very high.



(a) Using new TP initialization at time t_{change}: quick adaptation and convergence at $t = 1.05$

(b) Not using TP initialization at time t_{change}: slower adaptation and convergence at $t = 1.2$

Figure 4.6: The costs of multiple planner output trajectories $\boldsymbol{q}^i$ being optimized in parallel for one dynamic obstacle scenario. Between $t = 0$ and t_{change} all planner processes converge to good solutions. At $t_{\text{change}} = 0.9s$ all planner movement costs worsen when the obstacles change velocity, and new motion trajectories are required to go to the target without hitting an obstacle.

Table 4.1: Average costs for 200 simulations of dynamic obstacle reaching.

Initialization Method	2 Initializations at $t = 0$ and $t = t_{\text{change}}$	1 Initialization at $t = 0$
LINEAR-iLQG	0.70 ± 0.05	1.02 ± 0.06
LINEAR2-iLQG	0.55 ± 0.04	0.85 ± 0.05
TP1-iLQG	0.41 ± 0.03	0.9 ± 0.05
TP2-iLQG	0.39 ± 0.03	0.86 ± 0.05
TP3-iLQG	0.37 ± 0.03	0.82 ± 0.05
TP4-iLQG	$\mathbf{0.36 \pm 0.03}$	0.80 ± 0.05

If one looks again at Figure 4.3, it can be seen that if at time t_{change} the old best plan is not changed immediately, the robot will be hit by the obstacle. With TP a new plan avoiding the new path of the moving obstacle is generated almost immediately. The estimated costs of 4 different planner threads is illustrated in Figure 4.6. There it can be seen how all planners converge in the beginning, then the obstacle change velocities at t_{change} and the old plans become unfeasible. We show the estimated costs of the different motion trajectories $C(x; \boldsymbol{q}^i)$ which use the current information for the obstacle movement as the parallel framework proceeds in time. It can be seed how using an additional TP initialization at time $t = t_{change}$ improves greatly the reactive performance of the planners. The net gain of 0.15s faster planner convergence but it is crucial for performance in this scenario (even though for many static planning scenarios this would seem like a very short duration).

Table 4.2: Average costs for 200 simulations of sequential reaching, initialization method VS offline pre-optimization iterations.

# of offline iterations	$n = 0$	$n = 5$	$n = 10$	$n = 15$
LINEAR-iLQG	0.56 ± 0.06	0.52 ± 0.06	0.54 ± 0.06	0.54 ± 0.06
TP1-iLQG	0.22 ± 0.03	0.16 ± 0.02	0.14 ± 0.02	0.13 ± 0.01
TP2-iLQG	0.23 ± 0.03	0.15 ± 0.02	0.13 ± 0.01	0.12 ± 0.01
TP3-iLQG	0.20 ± 0.02	0.13 ± 0.01	0.12 ± 0.01	0.12 ± 0.01
TP4-iLQG	0.19 ± 0.02	0.12 ± 0.01	0.12 ± 0.01	**0.11 ± 0.00**

Results for Sequential Target Scenario in Simulation

Table 4.2 shows our results in the sequential target domain. First, we compared different initializations of the planners, with LINEAR being again the baseline, and TPn for n=1...4 being the TP initialization with 1, 2, 3 and 4 planner processes in addition to the control process. Second, we tested the gains from some offline pre-optimization before time $t = 0$. We indicate the use of n additional offline optimization iterations(each of which delays the movement start by $\tau = 0.07$s) in the columns of Table 4.2.

LINEAR fails to find good solutions consistently, even with extra pre-optimization iterations: a straight line path is difficult to be made into a curved detour path as in figure Figure 4.4(b). The results confirm that more parallel initializations are better for TP. More pre-optimization iterations lead to increased performance, mainly because they allow better transfer of the prototypes and better estimation of the costs required for the $argmin$ in line 4 of Algorithm 2. However, just a few iterations are enough to improve the trajectories before starting to follow, there is a negligible gain from doing more than 5 pre-optimization iterations at start time.

Hardware Demonstration

We also tested a sequential target setup with the Schunk LWA3 arm, depicted in Fig. 4.7. There we have a blue target ball moved by the human demonstrator around a cylinder obstacle. The human demonstrator deliberately changes the target location any time the robot arm comes near it. The online planner performed quickly and robustly in an interactive situation where a human moves the target[1].

We used an 8 core desktop computer. We used 2 cores for vision, 4 cores for iLQG optimization in parallel processes (with $T = 400$ and 4 seconds duration), 1 core for control. For vision we had a stereo Bumblebee camera and used the method of Linderoth et al. (2010) for calibration and learning to map 2D image coordinates to 3D world coordinates. We got a reasonable performance of 15 frames per second and average accuracy of 5mm.

[1] A video demonstrating the robot motions is available at `http://user.cs.tu-berlin.de/~jetchev/MultiPlan.html`

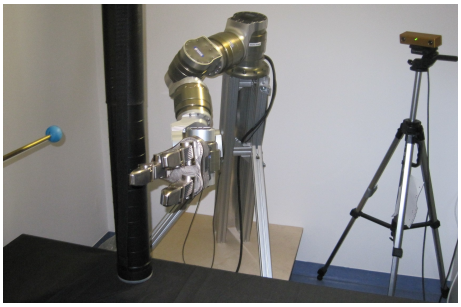


Figure 4.7: Our hardware setup: Schunk LWA3 arm, stereo Bumblebee camera, black obstacle and a blue target that the arm follows continuously.

4.6 Conclusions

We presented a parallel algorithm for online motion planning. Its main contributions are the utilization of multiple CPU cores for online planning in parallel, and the prediction of diverse initial trajectories that allow the system to explore and switch between alternatives depending on the environment. The last capability is achieved by using trajectory prediction for motion planning initialization. The machine learning approach of TP is applied to great effect to two challenging robot problems requiring quick reaction times to unpredictable changes in the world. The mapping learned from data between situations and motions codes some of the world structure in the predictor function. This structure is reused when encountering novel situations: the predictor initializes the planners to already reasonable motions and speeds up convergence. Our results showed how such exploration of multiple trajectories, made especially effective by splitting the computational load between multiple CPU cores, improves online planning performance: under time constraints our model manages to effectively generate feasible motions.

4.6.1 Future Work

Future work will look to apply the TP parallel framework to more realistic hardware robot scenarios. By selecting carefully the features available in the situation descriptor x and the task space used for prediction, the TPPF framework can be adapted to many possible scenarios. Right now the biggest hindrance to the further application of the TPPF to real scenarios with moving objects, is the sensor problem: how to get the most accurate information about the world in the fastest amount of time. Using an optical flow representation (Willert et al. (2007)) to capture motion structure for a whole scene without relying on individual object speed measurements can allow TPPF to reason about cluttered moving scenes.

It is also possible to incorporate the sensor uncertainty in the trajectory prediction module by adding error estimates to the features of the situation descriptor. We can then let the machine learning approaches used for TP to make sense of the features and their noise and learn appropriate mappings of situation to motion.

Chapter 5

TRIC: Task Space Retrieval Using Inverse Optimal Control

The next algorithm we developed is called *Task Space Retrieval Using Inverse Feedback Control* (TRIC). Its idea is to find simultaneously 3 aspects of the latent structure of motion data: a sparse discriminative value function specifying the desired motions, a motion representation of features relevant for a task, and a motion generation policy. To learn this we are using data in the form of demonstrated motion trajectories, without the strong assumption of knowing any cost function characterizing the observations and without knowing the task space in which the demonstrations should be executed. We demonstrated the effectiveness of our method by learning from imitation a controller for robot grasping of objects, a challenging high-dimensional task. TRIC learns the important control dimensions for the grasping task from a few example movements and is able to robustly approach and grasp objects in new situations.

5.1 Overview of the TRIC method

Learning complex skills by repeating and generalizing expert behavior is a fundamental problem in robotics. A common approach is learning from demonstration: given examples of correct motions, learn a policy mapping state to action consistent with the training data (DPL). However, the usual approaches do not answer the question of what are appropriate representations to generate motions for specific tasks. Inspired by Inverse Optimal Control, we present TRIC, first presented in our previous work Jetchev and Toussaint (2011a). The main contributions of TRIC for this thesis are:

- the discovery of a *sparse discriminative value function*
- the use of this value function to create a motion controller for *imitation* of the demonstrations in a robust and goal-oriented way
- the *analysis of motion features* relevant for a certain task

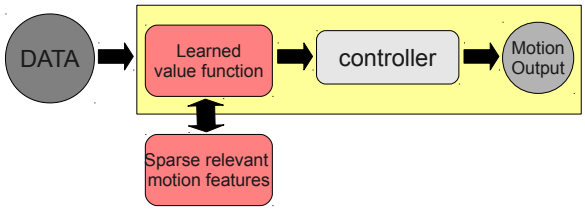


Figure 5.1: A diagram illustrating TRIC: use motion data to imitate observed movements. First learn a representation of the motions and a value function, then use the value function in a motion rate controller.

In Chapter 3 we assumed that we were given a cost function that specified what the desired motions were, and motion generation was just a matter of optimizing this cost. However, in general this cost function assumption does not hold, and we do not know the measure of a good motion for a task. Usually experts use their knowledge to design a cost function suitable for the desired motions, but this can be complicated and not intuitive for certain tasks. In some cases it is easier just to give examples of desired motions and construct a cost (negative reward) function from them (Ratliff et al. 2006). Note that the power of inverse optimal control methods is that without having a specific task description (e.g. in the form of a cost function), we can infer the desired behavior just from demonstration. Here we take a slightly different approach and learn a *value* function representing the desired behaviors.

Suppose we are given a set of motions and no additional information about them. We can ask the question *"Why were the motions performed in this way?"*. The implicit assumption we make is that the teacher that showed us this demonstrations is rational and the demonstrations were optimal with respect to some criteria defined in some motion feature subspace. Both the optimality criteria and the relevant motion features are not directly recorded in the observations. Both can be considered latent structure in the data, that can be eventually discovered by proper learning methods. The essence of our algorithm is to learn a value function f that characterizes the demonstrations. The teacher movements should always decrease f, which can be viewed also as a motion potential leading to correct motions (Howard et al. 2008). Without any prior knowledge about the task at hand, the controller using the learned value function can effectively generalize motion from few training examples to new situations.

The concept of value function as used by TRIC is related to the notion of value function in RL, but we are representing expected discounted costs (negative rewards) instead of discounted rewards. A value function has the following properties (see Section 2.4.2) that make it appealing to use for learning and generating motion:

- the value function in RL is a representation of the expected discounted return of a given policy
- given an optimal value function, the optimal policy is to maximize the immediate reward and expected future rewards

In relation to the first property, a high-level view of the value function f is that it represents the goal or intention of the teacher's policy that created the demonstrations. This is not a fixed description of a desired robot configuration, but a whole subspace of states. Generating motion with this subspace as a goal allows to satisfy the task in a flexible way, similar to the task manifold concept of Gienger et al. (2008). In relation to the second property, we can use the learned value function to generate motions with a reactive controller that goes smoothly in the direction of decreasing the value.

Another question we can ask is *"What are the task spaces in which this motion is performed best?"*. For example, if the teacher had the goal of reaching to a target object with his endeffector the task space Y_{target} of relevant coordinates in the target frame will be the best one. Usually experts would choose a task space and just assume that it is the proper one for a given task (see Section 2.4.1). It is difficult to design behavior for more challenging tasks like object manipulation, because the correct task spaces, targets and costs are not known apriori. Other approaches would take a set of task spaces and test which of them works best with respect to a certain criteria; see e.g. our approach to TP in Section 3.4.1, used also by Muehlig et al. (2009); Billard et al. (2004). However, in most interesting robotic setups there will be too many task spaces and the robot designer cannot test every single one of them to see which works best for some task. Learning a value function from teacher demonstrations has many advantages in such cases, and it can also give a satisfactory answer to the question of "What to imitate". The next central idea in our approach is to model f as a sparse discriminative value function of a high-dimensional feature vector y, which offers a large variety of potential geometric features that might be relevant for a motion. The optimization of f implies a choice of these features. Thus we answer the question of *What* in imitation learning in a principled way using data and Machine Learning by coupling the learning of value function characterizing motion data with task space retrieval.

Our approach is to define a large set of motion features from which we can extract a compact set of the important task dimensions using TRIC. The choice of features $\phi(q) = y$ effectively determines what value functions we can learn, and what task spaces we can select as relevant for a given task. However, this is a weak prior restriction, since with our method we can select the important dimensions out of any set of motion features and we can easily take as landmarks all objects in a scene, and any geometrical relations between them. As long as our feature set y is rich and contains multiple controllable task spaces of the robot body parts relevant for the task at hand, we can expect to successfully imitate with TRIC a large class of motion problems with such motion features. The concrete choice of features will be presented in Section 5.5.

In Figure 5.1 we illustrate the main concepts of TRIC, and in the next textbox we summarize what TRIC requires and what it can accomplish:

Premises necessary for the TRIC framework:

- a **controller** algorithm and a rich set of **motion features**
  "the ability to control the robot in different ways"
- ability to record **data**: situations (e.g. object positions) and motions
- an assumption that the **teacher** was **rational** and had a **latent goal**
  *"the **teacher** showed the desired behavior and there was a hidden criteria specifying why it is desired"*
- the world **situations** are sampled from a **generating distribution**
  "the objects in the world, their count, size and positions, have a certain pattern"
- we **limit** the class of behaviors we want to imitate to non-periodic motions

Given these premises, TRIC can **accomplish** the following:

- learn a **value function** for the task
  "learn the teacher's hidden goal that characterizes the observations"
- the value function leads to a **controller** to **imitate** and **generalize** the data
  "imitate in unseen situations from the same distribution"
- learn a **sparse** motion **representation**
  "understand what the teacher did in what task spaces"

This chapter will proceed to explain and demonstrate the TRIC algorithm. In Section 5.2 we describe our motion model, that is, how the value function f implies the movement. In Section 5.3 we state the desired properties of the value function f and define a training loss for learning f from demonstrations and explain the significance of each term. We will discuss some of the properties and limitations (i.e. why the motion data should be non-periodic) of TRIC in Section 5.4. We will test the effectiveness of our method in a robot grasping application and analyze the best features for grasping in Section 5.5. Finally, Section 5.6 concludes the chapter and gives some ideas for future work.

5.2 Motion Model and Controller

The method of TRIC was designed for generating motion with articulated robot models. We write the vector of joint angles of the robot as $q \in \mathbb{R}^n$ and a robot movement trajectory for T time steps is $\{q_i\}_{i=1}^T$. We assume that for each joint configuration q we can compute a high-dimensional feature vector $\phi(q) \in \mathbb{R}^s$, which we will describe in detail in Section 5.5. The feature vector will typically comprise all possible relative and absolute positions and distances between a set of landmarks defined on the robot and external objects – clearly, the size of the feature vector is quadratic in the number of landmarks.

It is the objective of learning to select relevant features to describe the value function f of a motion. The features $\phi(q)$ are non-linear in q and can be computed from the robot kinematics. E.g. if the task is to move the robot hand to a target position, a properly chosen feature that codes this distance will allow much easier control than the robot joint configuration space. TRIC uses the richness of the representation $\phi(q)$ to learn a value function model and select important features, see for illustration Figure 5.2.

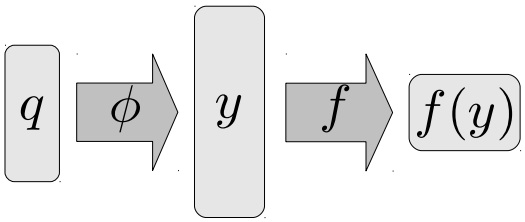


Figure 5.2: Motion representation scheme: the joints q are mapped by ϕ to a high-dimensional feature vector y, which is then used to find the value function $f(y)$. In general y has many more dimensions than q; f should be sparse in y.

Note that strictly speaking we are using for the features a task space projection $y = \phi_x(q)$ of the joint state q in some situation x containing the state of the world at this instance, as in Section 3.4. For simplicity in notation for TRIC we are skipping the situation subscript in ϕ_x, and it should be assumed that the motion features are always calculated with respect to the current world situation x.

We assumed that the teacher was rational with respect to some hidden criteria. Thus, if we suppose that f is trained as a value function consistent with the observations, the robot can generate motion with a method commonly used in robotics: motion rate control (see Section 2.2). More precisely, the robot motion is generated by imposing a smooth decreasing "motion" on f which is translated back to joint angle motions using inverse kinematics (IK).

More formally, motion rate control can be defined as computing a new robot pose given the old pose q_t in each time slice t by minimizing the next step function C^{MR}:

$$q_{t+1} = \underset{q}{\text{argmin}}\ C^{MR}(q, q_t) \tag{5.1}$$

The next-step cost C^{MR} is defined similarly to C^{IK} from Equation (3.8) in Section 3.4:

$$C^{\text{MR}}(q, q_t) = \underbrace{\|q - q_t\|^2}_{\text{small step}} + \underbrace{\|f \circ \phi(q) - f \circ \phi(q_t) + \delta_2\|^2_{\delta_1}}_{\text{decrease } f \circ \phi(q)} + \underbrace{C_{\text{prior}}(q)}_{\text{motion priors}} \tag{5.2}$$

The first term $\|q - q_t\|^2$ penalizes step length, the standard way to prioritize smooth movements. The second term $\|f \circ \phi(q) - f \circ \phi(q_t) + \delta_2\|^2_{\delta_1}$ aims for a decrease in the value function $f \circ \phi(q)$ by a rate δ_2. The constant δ_1 controls the importance of this term with respect to the other terms of C^{MR}. The third term $C_{\text{prior}}(q)$ imposes additional

standard costs for joint limits and collisions, i.e. our prior constraints on the motion to be generated.

Iteratively making steps $q_t, q_{t+1}, ...$ with this motion model generates a continuous motion trajectory decreasing the value function $f \circ \phi(q) = f(y)$. Note that our goal is not to find a joint posture having the global minima of the value function in one step, but to generate motion *smoothly decreasing* the value function. With our motion model (5.1) we do not aim to recover the exact sequence of joint states from the demonstrations, but learn to generate motions good with respect to the (latent) value function. We lose the speed information and relax the problem; the speed is tuned by the parameters in equation 5.2 which control how fast we decrease the value function. Because of the linearization used in IK (Section 2.2), we can only expect that IK makes small steps towards the goal.

The gradient of $f \circ \phi$ with respect to q can be found using the chain rule:

$$\mathcal{J} = \frac{\partial(f \circ \phi)}{\partial q} = \frac{\partial f}{\partial \phi}\frac{\partial \phi}{\partial q} \tag{5.3}$$

The first term $\dfrac{\partial f}{\partial \phi}$ can be calculated analytically given a parametrization of f. The second term $\dfrac{\partial \phi}{\partial q}$ is the task space Jacobian that can be calculated from our articulated robot simulator for a specific posture q. Because f is a function of task space feature inputs, it can be viewed in itself as a more complex constructed parametrized motion task space.

With this motion model we have the following property, whose proof can be found in the appendix.

Proposition 5.2.1 *If $\delta_1 \to \infty$ then the IK solution q_{t+1} minimizing Equation (5.2) has the property that the next step $q_{t+1} - q_t$ is proportional to the value function gradient $\mathcal{J}$.*

An alternative motion model can directly make steps proportional to the gradient of the cost-to-go Equation (5.2) and this will also minimize the value function and result in motion imitation. However, the motion rate control formulation is more robust with respect to satisfying different terms with different weights, and is preferred for robot controllers.

Note that even though we can add as many task variables as we want and use motion rate control, in practice having more variables reduces the performance of the algorithm. TRIC is designed to find a sparse set of task variables which are important for the specific task, which has these advantages:

- regularization of the value function prevents overfitting and improves the learning performance
- having a large set of task spaces means that probably some of them are just noise and irrelevant for the task at hand, so it is better to not follow blindly such noisy motion aspects
- having few task spaces as targets makes it more likely that IK will find joint states satisfying all of the task space targets with reasonable accuracy

5.3 Learning the TRIC Value Function from Motion Data

We defined in the previous section the motion model we use, assuming a value function f existed. Now we will explain how we can learn this value function f such that our motion model generates motions similar to demonstrated behaviors. We assume that f is a differentiable function parameterized by the vector w – in Section 5.5 we will specify f concretely.

Suppose the robot observes $D = \{q_t^i, y_t^i, \frac{\partial \phi}{\partial q}(q_t^i)\}_{i,t}^{N,T}$, a set of N demonstration trajectories of length T each. Our training algorithm can handle trajectory data of varying time lengths because the loss function sums over all trajectories and their time steps. We assume for simplicity that all trajectories are fixed to T time steps. The data consists of joint space trajectories movements, their projections to the motion feature space $\phi(q_t^i) = y_t^i$, and the Jacobian matrices of this feature space $\frac{\partial \phi}{\partial q}(q_t^i)$.[1] A dataset with sampled motion trajectories assumes indirectly that the demonstration steps go linearly between the samples q_t and q_{t+1}. We can call this the *keyframes* assumption. It is an indirect assumption behind the majority of DPL methods. It is up to the teacher providing demonstrations that the sampling rate is high enough to capture the essential demonstration properties.

The main assumption we make for learning a value function from data is that the teacher is rational and the demonstrations were optimal with respect to some criteria defined in some motion feature subspace. One way to incorporate this assumption is to design a value function f with the following 3 properties:

- (i) The demonstrations move to states of lower value as time progresses: f acts as a **generative** motion potential.
- (ii) Demonstrated teacher motions are discriminated against any other motions: **discrimination**.
- (iii) A small set of relevant motion features are selected: **sparsity**.

Property (i) arises from the assumption that a rational teacher moves closer to his desired goal as time progresses. This property is also fundamentally necessary for our motion model Equation (5.1) where we want to reproduce the teacher's motion by generating joint states with gradually lower value f.

Property (ii) arises from the fact that the motions of the rational teacher should have lower value function than any random motions, because they are closer to the goal of the demonstration than random motor commands.

Property (iii) comes from the assumption that some motion subspace has a good description of the latent goal of the teacher. To ensure that f has these 3 properties we

[1] If it is clear from the context that we are dealing with just one trajectory, we will skip the superscript i and write just q_t instead of q_t^i.

define the training loss $L(D; w)$ for data D and value function parameters w:

$$L(D; w) = \sum_{i=1}^{N} \sum_{t=1}^{T} (\alpha_n L_n(y_t^i; w) + \alpha_g L_g(q_t^i; w)) + \alpha_w |w|_1 \tag{5.4}$$

The hyperparameters $\alpha = \{\alpha_n, \alpha_g, \alpha_w\}$ determine the influence of the different loss terms. Once we have defined the loss $L(D; w)$ we can use gradient-based optimization to optimize with respect to the parameters w to learn the function f. The loss is related to the 3 properties of TRIC in this way:

- Both loss terms L_n and L_g influence property (i).
- Property (ii) is ensured directly by the design of the loss term L_n, but it is also influenced by the term L_g.
- The L_1 regularization term $|w|_1$ in Equation (5.4) forces sparsity in the parameters of the function $f(y)$ and indirectly performs feature selection with respect to y. Because of the coupling $f(y) = f \circ \phi(q)$, the sparsity of w means that the motion rate control will lead to joint states q changing some task dimensions of y and not caring about others – which we call task space retrieval and selection, property (iii).

The next sections will explain in more detail L_n and L_g, and how they influence the value function f and ensure that properties (i) and (ii) hold. In appendix A.5 we write the exact computational complexity of training TRIC and generating motion using f.

5.3.1 Discrimination in Feature Space via Loss Term L_n

We want to design the term loss L_n so that properties (i) and (ii) hold. First we make the following observations:

- The demonstrated trajectories were assumed to be near optimal and should have low values f.
- All other possible motions should be discriminated from the true motions and have high values f.
- The trajectory samples ahead in time should have lower f values than those before them.

We use an approximation to model the relation between the teacher demonstration and what we called informally *all other possible motions* or *noise*. We create a small set of $m = 1, .., M$ of synthetic *noisy* samples which need to be discriminated from the demonstrated trajectories:

$$\tilde{q}_{t,m}^i = q_{t-1}^i + \mathcal{N}(0, \sigma^2) \tag{5.5}$$

$$\tilde{y}_{t,m}^i = \phi(\tilde{q}_{t,m}) \tag{5.6}$$

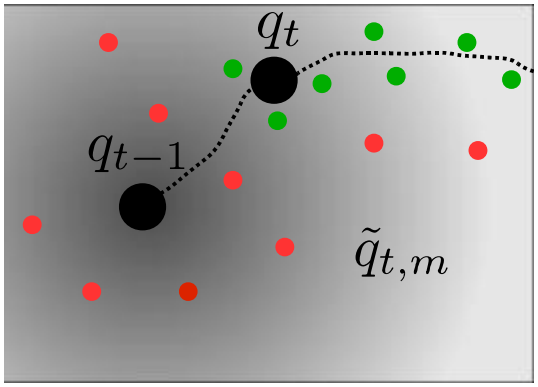


Figure 5.3: The noise $\tilde{q}_t$ sampled from a Gaussian centered at q_{t-1}. A few *noisy* samples are shown, colored by their weights $\epsilon_{t,m}$. Low weights are green and high weights are red (for visual clarity the weights are discretized to 2 colors).

The noisy joint states $\tilde{q}_{t,m}$ are created by adding Gaussian noise with variance σ^2 to the robot joint configuration of the previous time slice q_{t-1}. We add noise on q and not on y directly since the feature vectors y lie on a subspace constrained by the robot kinematics and dependencies between the dimensions of y. Thus, we sample in joint space resulting in task space samples lying on a low-dimensional subspace $Y^{\text{valid}} \subset \mathbb{R}^s$ where s is the number of dimensions of y. If we have n joints, this would be a no more than n-dimensional subspace.

We define the distance between a joint state and a joint trajectory interpolated linearly between its keyframes:

$$d_{\text{curve}}(q, \{q_\gamma\}_{\gamma=t}^{T}) = \min_{\lambda \in [0,1], \gamma \in [t,T]} \|q - q_\gamma - \lambda(q_{\gamma+1} - q_\gamma)\| \tag{5.7}$$

We define sample weights for a sample $\tilde{q}_{t,m}^i$ equal to the distance to the true demonstration trajectory *ahead* in time:

$$\epsilon_{t,m}^i = d_{\text{curve}}(\tilde{q}_{t,m}^i, \{q_\gamma^i\}_{\gamma=t}^{T}) \tag{5.8}$$

Samples that are near a correct state q_t in the future will have low weights. Samples that are away from the true trajectory in the future will have high weights and their loss contribution will be higher. Such sample weighting represents this prior for metric in the space of $y = \phi(q)$, and is also consistent with the keyframe assumption we made in the previous section 5.3.

The way we create these synthetic samples and weight them is illustrated in Figure 5.3. The term L_n is then defined as:

$$L_n(y_t^i; w) = \sum_{m=1}^{M} \epsilon_{t,m}^i \log(1 + e^{f(y_t^i; w) - f(\tilde{y}_{t,m}^i; w)}) \tag{5.9}$$

We use the log loss from Equation (2.12), common in discriminative learning, to penalize noisy samples with low value f.

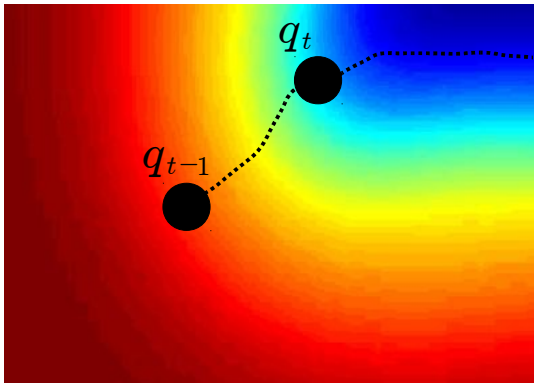


Figure 5.4: The loss term L_n pushes down the value function $f \circ \phi(q)$ near the true trajectory samples q_t , and up the cost of the generated noise. Red represents high value.

Minimizing the loss term L_n directly ensures the **discriminative** property (ii). An intuitive interpretation is that we "push down" the value $f(y)$ of the true samples and "push up" the value of the artificial noisy samples, proportional to their weights as in Figure 5.4, where for simplicity $q = y \in \mathbb{R}^2$.

Minimizing the loss L_n ensures that the **generative** property (i) holds under certain assumptions, as the following lemma shows.

Lemma 5.3.1 *Assume that the following holds for TRIC and the training demonstrations:*
(i) a single linear trajectory $\{q_t\}_{t=1}^T$ *in the joint task space* $\phi(q) = q \in \mathbb{R}^n$,
(ii) $M \to \infty$ *noisy samples,*
(iii) $\text{sign}(a-b)$ *is used instead of* $\log(1+e^{a-b})$ *in the terms of* L_n *in Equation (5.9),*
(iv) values of the metric d_{curve} *in Equation (5.7) smaller than a constant* ϵ_0 *are set to 0.*
Then a lower bound L_B *on the loss* $L' = \sum_{t=1}^T L_n(q_t; w)$ *exists, and when* $L' = L_B$ *it holds for all time steps* t *that (I)* $f(q_{t-1}) > f(q_t)$*, i.e. the value function is decreasing along the trajectory, and (II)* $f(q_T)$ *is a minimum of the value function.*

The proof of this lemma is in the appendix.

Properties of the Attractors of f

The different minima of f act as attractors to the motion generation system defined in Equation 5.1. We can analyze its stability properties using notions from the classical Lyapunov Stability theory (Slotine and Li 1991), redefined for a motion rate control system instead of the usual differential equation notation $\dot{x} = g(x)$. If a motion system converges to some equilibrium point regardless of the starting point it will be called asymptotically stable.

Definition 5.3.2 *The motion system* $x_{t+1} = x_t + g(x_t)$ *is asymptotically stable at the point* x^* *if starting from any* x_0 *it converges asymptotically to* x^**, i.e.*

$$\lim_{t \to \infty} x_t = x^* \tag{5.10}$$

Proving such a property for a motion system is useful, because it signifies that the motions can reach their targets robustly with respect to any perturbations and random starting conditions (Perkins and Barto 2002; Khansari-Zadeh and Billard 2010). There is the standard Lyapunov Stability theorem that states a condition when the asymptotic stability property holds:

Theorem 5.3.3 *The motion system $x_{t+1} = x_t + g(x_t)$ is asymptotically stable at the point x^* if a continuous and continuously differentiable function $V(x)$ can be found such that:*

$$\begin{cases} (a) & V(x) > 0 \quad \forall x \neq x^* \\ (b) & V(x^*) = 0 \\ (c) & \dot{V}(x) < 0 \quad \forall x \neq x^* \\ (d) & \dot{V}(x^*) = 0 \end{cases} \tag{5.11}$$

Now we proceed to claim that TRIC is asymptotically stable under some assumptions, and the proof is in the appendix.

Proposition 5.3.4 *Suppose we have trained TRIC on a single trajectory $\{q_t, y_t\}_{t=1}^T$ and that the implications of Lemma 5.3.1 hold for a trajectory in that task space. Additionally, we generate motion with $\delta_1 \to \infty$, i.e. very high weighting of the value function. Then the motion generated by the model in Equation (5.1) fulfills the conditions of Theorem 5.3.3 and is thus asymptotically stable in the joint space subspace $Q' = \{q' : \phi(q') = \phi(q_T) = y_T\}$.*

This proposition tells us that we have as attractor a whole joint subspace Q' to which our motion generation method will converge given enough time. The attractor is not a single state but a subspace, because redundancy of the joint space allows for multiple different robot joint configurations q' to have the same task features y_t.

It is not trivial to prove convergence properties of TRIC in a realistic scenario with multiple trajectories, limited samples M and δ_1 not going to infinity. Our experiments in Section 5.5 show empirically that motion generation using the learned value function is robust and stable. However, it is obvious that if we have extreme cases, e.g. an obstacle blocking all paths to to the target and pushing the robot away, no performance of the system can be guaranteed because obstacle avoidance will influence the behavior more than the task-related value function.

5.3.2 Making the Gradient Consistent with the Demonstrations via Loss Term L_g

The next loss term we discuss is L_g. Its main idea is to force the negative gradient $-\mathcal{J}$ to point in the direction of difference vector (i.e. velocity) between steps $q_{t+1} - q_t$ of the demonstrated trajectories in joint space. There are several reasons for this design:

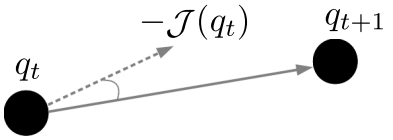


Figure 5.5: A trajectory sampled in q_t and q_t+1. The loss term L_g is minimized when the negative gradient $-\mathcal{J}(q_t)$ has maximal cosine of its angle with the direction of the next step $q_{t+1} - q_t$.

- The teacher that gave the demonstrations was assumed to be rational, so we expect that he moved in the direction of steepest value descent between q_t and q_{t+1}. The *keyframes* assumption we made in Section 5.3 ensures that the trajectories are approximately linear between the samples in time available in the dataset.
- According to (5.2.1) the solution for optimal next step q_t of the motion rate control model has the property that our controller creates steps in joint space proportional to the negative gradient $-\mathcal{J}$.
- We want to ensure that properties (i) and (ii) hold.

The loss term L_g is defined as:

$$L_g(q_t; w) = \frac{\mathcal{J}(q_t)^T}{\|\mathcal{J}(q_t)\|} \frac{(q_{t+1} - q_t)}{\|q_{t+1} - q_t\|} \tag{5.12}$$

where $\mathcal{J}(q_t)^T$ is the gradient of $f \circ \phi$ evaluated at this particular joint state q_t. Minimizing the cosine of the angle between $\mathcal{J}(q_t)$ and $q_{t+1} - q_t$ is equivalent to maximizing the cosine of the angle between $-\mathcal{J}(q_t)$ and $q_{t+1} - q_t$, which is maximal when these are parallel and have angle 0. Thus we can see that minimizing loss L_g has the effect that $-\mathcal{J}(q_t)$ points in the direction of the next step $q_{t+1} - q_t$ as in Figure 5.5.

This effect and the mathematical properties of the gradient (steepest directional derivative) determine the relation of the term L_g to properties (i) and (ii):

- The value function f decreases along the vector $q_{t+1} - q_t$, thus f decreases in time – related to the **generative** property (i).
- For a local neighborhood $Q_\gamma := \{q\}$ for which $\|q - q_t\| < \gamma$ for some small enough γ it holds that
$$f \circ \phi(q_t - \gamma \frac{\mathcal{J}(q_t)}{\|\mathcal{J}(q_t)\|}) < f \circ \phi(q) \qquad \forall q \in Q_\gamma$$
This is discrimination between the joint state on the (linearly interpolated) true teacher's trajectory $(q_t - \gamma \frac{\mathcal{J}(q_t)}{\|\mathcal{J}(q_t)\|})$ and any other joint states in the subset Q_γ – related to the **discriminative** property (ii) for this specific local neighborhood.

Note that computing $\dfrac{\partial L_g}{\partial w}$ involves calculating

$$\frac{\partial \mathcal{J}}{\partial w} = \frac{\partial^2 f}{\partial \phi \partial w} \frac{\partial \phi}{\partial q} \tag{5.13}$$

The first term on the right side can be calculated from f analytically, and the second is the so-called kinematic Jacobian, which we also have in our training data D. This means that minimizing L_g can be done offline, without simulator calls, which simplifies the training architecture and improves performance.

On Why the Term L_g is Defined in Joint Space

Note that it is reasonable to use the term L_g to condition where the derivative of the value function points with respect to *joint space* observations q and not *feature space* observations $y = \phi(q)$. The joint values are in a relatively low dimensional space, and we expect that only joint values directly relevant for the task goal were moved, otherwise they would have stayed in place because of the "laziness principle" for good motions, see 2.2. The y observations are high dimensional and with many possibly irrelevant dimensions, so it is not reasonable to expect that these dimensions all change in a steepest descent manner. For example, suppose that the real value function only depends on the first dimension of y, which we denote $y_{[1]}$. Changing just this $y_{[1]}$ would be the most rational way to lower the value function. However, all dimensions of $y = \phi(q)$ are coupled because of the robot structure, so it can be that in the observed trajectories some irrelevant dimension $y_{[2]}$ was also changed. It would be undesirable to condition the value function and its derivative to change $y_{[2]}$ as observed because this dimension is just noise.

5.4 Discussion of TRIC

In this section we will first discuss the motion imitation qualities of TRIC and then its limitations. Afterwards we will go in more detail and discuss the loss terms L_n and L_g and how they relate to established methods for imitating motion.

5.4.1 Local and Global Imitation

The above motion model creates a system capable of imitating behavior in a different way from both the DPL and IOC methods we mentioned in Section 2.4. We can see DPL, IOC and TRIC as three different conceptual approaches to imitation, using the concepts defined in Call and Carpenter (2002):

- DPL would imitate by memorizing an action (output) to execute given a state (input situation). It would *repeat* an action without regard of potential outcome – *mimicry*. Such an approach *cannot adapt* already seen motions to new unseen situations, so for good performance one would need to provide examples of every possible state and the action for it. If we use more interesting representations for input state and action one can probably get more interesting behavior, but with more features learning a mapping would be more difficult.

- IOC would learn rewards (negative costs) for each state consistent with a rational teacher. A global planner is needed to generate a motion (sequence of states) that maximizes the *immediate* rewards obtained in each state. Because the rewards reflect *local* properties of states, the planner can generate motion paths that differ from the training sequences, but are optimal with respect to the same rewards – *goal emulation*.

- TRIC learns a value function, such that going to states of low value imitates the desired behavior. This value function represents aspects of the *expected* future costs (negative rewards). We can interpret that the value function is a "representation" of the *global* goal (low value), but this value function is also trained to imitate the observations whenever possible. This would be *imitation of both goal and action* in the definitions of Call and Carpenter (2002).

In TRIC the learned value function $f : \phi(q) \mapsto \mathbb{R}$, or equivalently $f \circ \phi : q \mapsto \mathbb{R}$, determines the motion (policy) of the system, a role analogous to the rewards in IOC and Reinforcement Learning, see Section 2.4.2. However, unlike IOC where an immediate reward is learned, we learn a value function f in the TRIC model, corresponding to expected future reward. This means that another difference with IOC is that we do not need a global planner to create a whole motion maximizing the sum of rewards for each step. We can use a local controller to follow this value function, e.g. by using motion rate control with Equation (5.1). The value function f reflects a more global aspect of motion, similar to expected future rewards in RL. The way we learn f (Section 5.3) ensures that decreasing f via the term $\|f \circ \phi(q) - f \circ \phi(q_t) + \delta_2\|^2_{\delta_1}$ in Equation (5.2) leads to motion similar to the whole teacher trajectory. The other terms $\|q - q_t\|^2$ and $C_{\text{prior}}(q)$ in Equation (5.2) correspond to priors about the next motion step (e.g. smooth change and no collisions), playing the role of an immediate reward.

5.4.2 Limitations of the TRIC Model: Periodic Motion

The value function f acts as a potential over states $\phi(q_t)$ and TRIC is designed with the premises that (I) f decreases monotonically across the trajectory as it goes ahead in time and that (II) f has some minima to which the system goes and stops. We proved that TRIC is consistent with (I) and (II) using the assumption that the trajectories are lines or paths that do not cross themselves, see Lemma 5.3.1. This assumption deliberately excludes motion trajectories that cross themselves, which is often the case in tasks that have a looping periodic aspect. Such tasks cannot be solved by an attractor to a subspace of low value. An example of this issue can be following a demonstrated circular trajectory of the robot hand around a central point. The teacher makes two complete cycles around the center, and expects the robot to understand this behavior and learn to loop in such circular trajectories. Suppose the robot starts at some point q_0 on the circle. The TRIC value function $f \circ \phi(q)$ would need to decrease constantly around the edge of the circle as the robot moves and the motion controller always goes in the direction of decreasing value function. Once we make a full cycle and go back to the initial position q_0, the value

function $f \circ \phi(q_0)$ will have to be lower than the initial value $f \circ \phi(q_0)$. This is clearly a contradiction, and indicates a design limitation of the current TRIC model. Another approach to modeling the value function would be to add the last state of the robot q_{t-1} as information and learn a value function over data $f \circ \phi(q_{t-1}, q_t)$. This is a potential solution to the circular movement problem indicated above, but it is left to future work.

5.4.3 Discussion of the Loss Terms

Discussion of the Loss Term L_n

The L_n Equation (5.9) has similarities with IOC and Equation (2.8):

- the expected state features $\mu(\pi)$ correspond to our motion features y
- the comparison of true policy π^* to the other policies π is similar to comparing the true trajectory sample y_t to the noisy data points $\tilde{y}_{t,m}$: both lead to terms in the training loss that are large when the true trajectory is not preferred to any other motion
- the scalable margin $\mathcal{L}(\pi^*, \pi)$ corresponds to the weights $\epsilon_{t,m}$: both quantify how large the penalization in the training loss terms is: larger when a preferred false trajectory is away from the true trajectory

However, a difference is that the rewards in IOC are *immediate* rewards of next step transitions, whereas the value function f signifies rather an *expected* future cost (negative reward), see Section 5.4.1.

A comparison can also be made between discriminative learning in ML and the term L_n. The usual approach to discriminative learning as in Equation (2.11) would require finding the most offending false answer and discriminating it from the true answer. We do a similar discrimination (where value function corresponds to energy) between the true sample y_t and the noisy data points $\tilde{y}_t$. However, we use a discrete set of such samples to approximate the space of these samples in a small neighborhood of the last joint state q_{t-1}. This is much faster than looking for the most offending false answer, but potentially inaccurate if there are not enough samples. Another difference is that we do not have input vectors x as in the classic discriminative learning: we discriminate just in the space of outputs y.

Discussion of the Loss Term L_g

It can be shown that L_g is related to Howard et al. (2009) and DPL as formulated in Equation (2.6). Consider the normalized gradient $\frac{\mathcal{J}(q_t)^T}{|\mathcal{J}(q_t^i)|}$ as control policy $\pi(q_t)$ and the control signals $u_t = \frac{q_{t+1}-q_t}{|q_{t+1}-q_t|}$, normalized to constant length. Then both the DPL loss in Equation (2.6) and L_g loss in Equation (5.12) will be minimized when $\pi(q_t)^T u_t = 1$ for all joint states q_t. When policies and motor commands are normalized, minimizing

projection discrepancy becomes equivalent to minimizing the angle between motor command and policy prediction. The relation of DPL and L_g is also a strong indication that the term L_g leads to a value function robust to unseen constraints, just as the method of Howard et al. (2009), but we have not investigated this in detail.

Another analogy can be made between the TRIC loss with terms L_g and L_n and the hybrid heuristic IOC approach of Ratliff et al. (2009a) that also combines two terms:

- discriminate the teacher's policy by giving it higher rewards – analogous to the role of L_n for TRIC, with low value function instead of high reward
- predict directly the next step using the previous state as input, similar to a DPL approach – L_g is related to DPL in the formulation of Howard et al. (2009), as we mentioned already in this section

Khansari-Zadeh and Billard (2010) present an error function for imitation learning with two terms: the difference between magnitudes of velocity vectors of the demonstration data and controller prediction, and the difference between the directions of these velocity vectors. Our loss term L_g is maximizing the cosine of the angle between the negative gradient $-\mathcal{J}$ and the velocity vector $q_t - q_{t-1}$ observed in training data. In Proposition (5.2.1) we showed how the gradient $\mathcal{J}$ is influencing the direction of the joint steps created by the motion rate controller, and thus the direction of the observed velocity vector influences the motion created by our model. However, the velocity magnitude information is lost by TRIC because of an additional scaling factor in our motion controller, see Proposition (5.2.1).

5.5 Experiments

The task we will examine in detail in order to demonstrate the performance of TRIC is grasping of objects by a robot. This section will present our findings when applying TRIC to grasping data. It is organized in the following way:

- In Section 5.5.1 we describe our robot experimental setup and why the grasping task is interesting for imitation learning.
- In Section 5.5.2 we define the parametric model used for the implementation of the value function f. We also define the rich set of motion features y and the settings of the motion rate controller.
- In Section 5.5.3 we present experiments and results comparing the average performance of TRIC on random grasping targets, compared to the teacher and DPL as baselines. We also look at the effect of different hyperparameters and settings, and examine a more complicated scenario: cylinder grasping in the presence of obstacles.
- In Section 5.5.4 we analyze the **discriminative** and **generative** aspects of TRIC on the trajectory dataset D, related to properties (i) and (ii) from Section 5.3. We also show the subspace of grasping motions revealed by analysis of the value function.

- In Section 5.5.5 we analyze the extracted **sparse** motion features, property (iii).

5.5.1 The Grasping Task

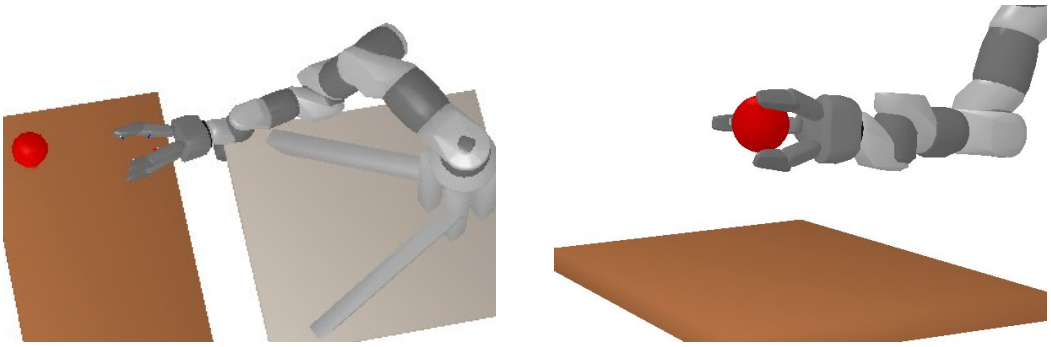
Motivation for Selecting Grasping for Experimental Testing of TRIC

Grasping is essential for robot manipulation of everyday objects. It is a complex manipulation task that requires control of multiple robot body parts with respect to an object. This means that there are many possible task spaces relevant for control. Grasping is a non-trivial task, which has received attention in the context of learning by demonstration and robotics. It is also a good example of how many different motion representations have been designed by experts for grasping. Tegin et al. (2009) acquire grasps from human movements and repeats them in joint space. Kroemer et al. (2009) explore grasps in a parametrized 6D space of position and orientation of the gripper, and learns a value function with regression. Kroemer et al. (2010) use the hand fingertips for grasping within the Dynamic Motion Primitives framework. Gienger et al. (2008) create a task manifold of presampled grasp positions and orientations and a motion potential towards them. None of the above works try to extract from data a representation for grasping. It is an interesting challenge to extract the important task spaces from data and learn good controllers with as few prior assumptions as possible for good grasping, and this is what we will demonstrate with TRIC.

Setup of the Grasping Task

Our robot has a joint configuration space $q \in \mathbb{R}^{14}$, as in the experiments in Section 3.7.3. We use as demonstration source the method of Dragiev et al. (2011), which is an efficient human-designed controller to grasp objects, with predefined reach and close fingers phases, as shown in Figure 5.6. The setup consists of the robot and a grasp target. We restrict the grasp target to be a sphere or cylinder to simplify the setup, but even with such simple geometry the grasp task requires complex robot motion. The controller of Dragiev et al. (2011) has several hard-coded phases, each of which controls different robot body parts and gives them different relative importance. All this information is hidden from TRIC, which has to recover the behavior just from observation. We visualize the joint angles for one sphere grasping trajectory in Figure 5.7.

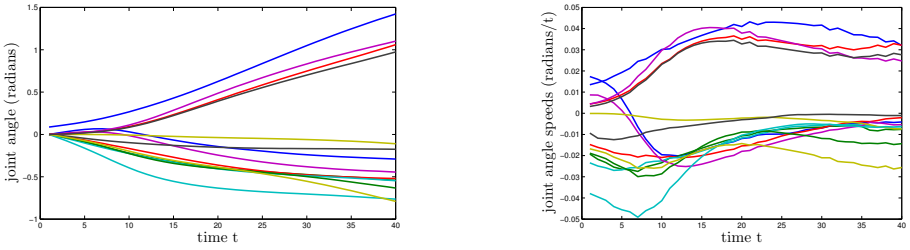
We use the errors of the task variables defined in Dragiev et al. (2011) as a cost metric to validate how good are the grasps of the TRIC controller (it did not play any role in training other than the fact that the teacher demonstrations were good with respect to this cost). The grasp cost is active only at the final grasping posture: it requires that all fingers lie at the object surface and are aligned to it, similar to Equation (3.31). This metric evaluates the final grasping posture of motion with a number between 0 and 1, where values below 0.25 are good grasps.



(a) First approach the red target object and open the fingers

(b) Finally align the fingers close to the object's surface

Figure 5.6: Illustration of a grasping motion that the robot has to learn.



(a) Absolute joint values in radians.

(b) Joint velocity values measured as $q_{t+1} - q_t$.

Figure 5.7: A sphere grasping trajectory in joint space for $T = 40$ time steps.

5.5.2 TRIC Setup: Motion Representation and Value Function Model

Selecting a Rich Set of Task Spaces

In Section 5.1 we said that the effectiveness of TRIC depends on selecting a rich set of task space motion features that seem reasonable for a certain task. For our experiments we decided to use a rich geometrical representation, namely the pairwise distances between a set of important landmarks, defined on the robot body and objects in the environment (similar to Section 3.3.1). The feature space $y = \phi(q)$ consists of $\hat{b}$ pairwise landmark relative positions p_i and their norms $d_i = \|p_i\|$ for each landmark pair $i = (j_1, j_2)$:

$$y = (p_1, d_1, ..., p_{\hat{b}}, d_{\hat{b}}) \in \mathbb{R}^{4\hat{b}} \tag{5.14}$$

Our approach to task space selection needs to define the set of landmarks and the geometric motion features, representing a prior on motion representations which are potentially suitable for the task at hand.

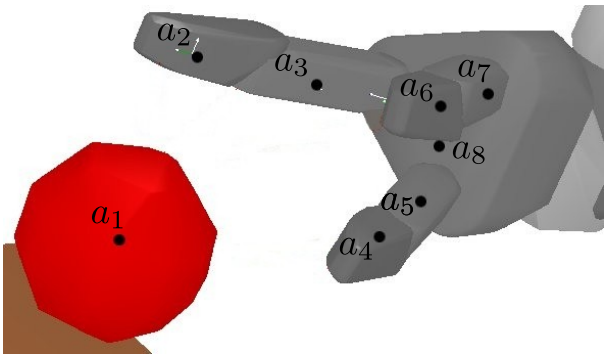


Figure 5.8: The features y are defined using the landmark set $A = \{a_i\}_{i=1}^{8}$, indicated by black dots.

Concretely for the grasping experiments we defined a set of 8 landmarks on the robot and the grasp target object $A = \{a_i\}_{i=1}^{8}$, shown in Figure 5.8. This includes the center of the target object a_1, the three fingertips a_2, a_3, a_4, the three lower parts of the fingers a_5, a_6, a_7, and the palm center a_8. The feature space consists of pairwise landmark positions and their norms: $y = \phi(q) \in \mathbb{R}^{224}$, as defined in the motion features Equation (5.14). We preprocess each dimension of y by rescaling it in $[0, 1]$. Some features are redundant due to the specific geometry of the landmarks and feature construction. First, $d_{i,j} = d_{j,i}$ because the norm of a vector does not change as it is rotated in some frame. Second the landmarks on fingers are defined on rigid bodies connected by joints, so their movement relative to one another is highly constrained, e.g. landmark a_2 always lies on the boundary of a 2D circle relative to the 3D frame of a_3. We remove the redundant features using correlation as a measure (Haindl et al. 2006), resulting in final motion features $y \in \mathbb{R}^{150}$.

This approach to modeling task spaces with information between geometric bodies in a scene is related to the interaction mesh approach in animation, see Ho et al. (2010). The authors use graph theoretic notions to animate and repeat motions in a Laplacian representation, i.e. each point is represented with respect to the average point of some neighbors on the graph. This leads to a set of motion constraints for each time slice of trajectory of fixed length extracted from a set of motion demonstrations, and they can be used to generate new motion imitating the demonstrated trajectories. In the case of TRIC we learn a value function leading to some goal-directed behavior, a more complex problem than simply replaying animation with small adaptations, but the representational power of our motion features is similarly based on the geometric information available in pairs of objects.

Parametrization of the Value Function

The description of the TRIC algorithm so far assumed that the value function f is a differentiable function parametrized by a vector w. Concretely for our experiments we

chose a sigmoid neural network with K nodes *plus* a linear term:

$$z = \frac{1}{1 + e^{W_1 y}} \tag{5.15}$$

$$f(y; w) = z^T W_2 + y^T W_3 \tag{5.16}$$

The parameters w consist of $W_1 \in \mathbb{R}^{K \times s}$, $W_2 \in \mathbb{R}^K$, $W_3 \in \mathbb{R}^K$ and $z \in \mathbb{R}^K$ is the hidden layer; $s = 150$ is the dimensionality of y.

The complexity of this neural model is $O(sK)$ for evaluation of $f(y)$, $\frac{\partial f}{\partial w}$ and $\frac{\partial f}{\partial y}$. The second partial derivative $\frac{\partial^2 f}{\partial y \partial w}$ has complexity $O(s^2 K)$, which depends quadratically on s. [2] We trained our models for 100 iterations with the BFGS method from the MATLAB Optimization Toolbox. The training time scales with K: for a linear model $K = 0$ it takes less than a minute, and 30 minutes for $K = 30$. We used Equation (5.1) for motion rate control using the learned costs $f \circ \phi(q)$.

Parameter Settings of the Motion Rate Controller

We rescaled the values of f so that the global minimum value of the model is -1. We did this by dividing W_2 and W_3 by $|W_2| + |W_3|$. This rescaling does not change the motion implied by the cost function, and allows for better comparison of differently trained models. Without such normalization the magnitudes of the value function f can be quite different, and this directly influences the motion generation Equation (5.2) and the number of steps required to reach the value function minima. By normalizing the value function after training we ensure that with one single setting of δ_2 the different value functions trained on the same dataset will have similar effect on the motion generation speed of TRIC.

We set δ_1 to 1000 and δ_2 to 0.03, which is a reasonable motion rate, usually requiring around 500 steps to reach the target. The speed of execution can be increased by taking a larger parameter δ_2. We used also the standard additional constraints on collision avoidance and joint limit avoidance coded in the term C_{prior}, as defined in Section 3.7, with weights high enough to ensure that the IK controller stays collision free and within joint limits.

Note that the coupling of different terms in Equation (5.1) and the multiple nonlinear task spaces incorporated in $y = \phi(q)$ create a quite complex problem for the IK method that uses linearization. It happens rarely that a single step will decrease the value function exactly by δ_2 , but the slower gradual decrease we get works well in practice.

5.5.3 Experimental Results: Performance of the Learned Grasping Controller

We examined grasping of a sphere and a cylinder. For quantitative comparison we inspected grasp costs of TRIC with different settings. For qualitative comparison we man-

[2] Here we write $\frac{\partial f}{\partial y}$ instead of $\frac{\partial f}{\partial \phi}$, because $y = \phi(q)$.

ually inspected whether the grasps generated by TRIC look plausible, and whether TRIC could generalize well to collisions with obstacles and grasp flexibly on different positions of the target object.

The default values of the hyper-parameters from Equation (5.4) are $\alpha_n = 1, \alpha_g = 1, \alpha_w = 0.001$, and the other parameters are $M = 60$ samples, $N = 27$ trajectories, $K = 30$ sigmoids for the model of f. We generate the M random samples by randomly sampling collision-free joint configurations with a normal distribution with $\sigma = 0.05$, corresponding to 0.05 radians joint angle standard deviation.

For training we generate a dataset by translating the position of the target object a_1 in a regular grid $50 \times 40 \times 40$cm in front of the robot, taking 3 positions in each grid dimension, leading to $N = 27$ different settings and training trajectories. Each grasp movement is generated for 5 seconds, and we sampled $T = 40$ time steps from it, once every 125ms. The target is grasped faster when the object is closer to the robot, so some of these motion trajectories grasped the object sooner or later than others.

Experiment 1 - Influence of the Hyperparameters on Sphere Grasping

For testing the grasp costs of the controller learned with TRIC we created a new set of 15 trajectories on random positions within the training grid and evaluated the grasping costs of different TRIC models with 500 steps taking 5 seconds for execution. Thus TRIC made one step every 10ms. This is a faster control rate than the sampling rate of our training set. This is due to the fact that we relaxed following the exact steps of the teacher and added the speed of imitation as a parameter to tune in our model (δ_2).

We tested 3 variations of the TRIC model by changing the parameters α_g and α_n controlling the weights of loss terms L_g (gradient direction) and L_n (discrimination from noise samples).

Figure 5.9(a) shows how the performance of TRIC changes with respect to the number of noisy samples M. Figure 5.9(b) shows how the number of test trajectories N influence the performance. We make the following observations:

- More samples M allow better training of TRIC, and having too few samples worsens performance. With 60 samples the performance is quite good (comparable to the teacher's) and we expect no further gains by increasing M to larger values. It seems that 60 samples were enough to approximate the local neighborhoods in the subspace of valid robot configurations (see Section 5.3.1).

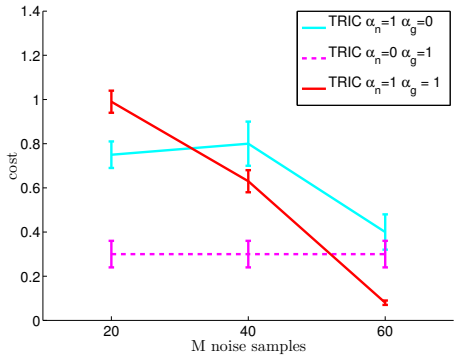
- With as few as 3 trajectories TRIC could learn grasping just as good as with 27 trajectories - efficiency in terms of N.

- The parameter M seems more critical than N. Since the training of TRIC is linear in terms of M and N, this shows that to speed-up training decreasing N is better than decreasing M.

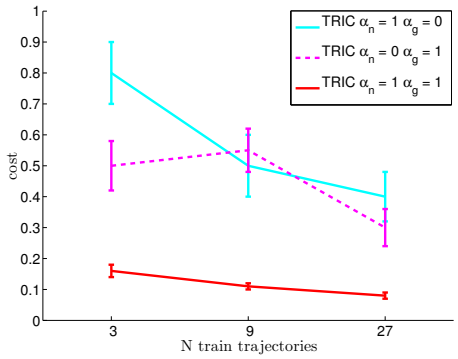
- The best hyperparameters of TRIC are $\alpha_g = 1$ and $\alpha_n = 1$, indicating that both loss terms L_g and L_n are important to learn the value function.

Table 5.1: Sphere grasping results: TRIC compared to the teacher on 15 test situations.

method	standard deviation and average of grasping cost
Teacher	0.06 ± 0.01
TRIC	0.08 ± 0.01



(a) The effect of changing the number of noisy samples M on TRIC.



(b) The effect of changing the number of training motions N on TRIC.

Figure 5.9: The effects of changing training setup parameters on TRIC: noisy samples M and training trajectories N are varied.

Overall, TRIC performed sufficiently close to the teacher heuristic from Dragiev et al. (2011), as we summarize in Table 5.1.

Experiment 2 - Comparison of TRIC with DPL in Sphere Grasping

In a second experiment we implemented as baseline the DPL method from Section 2.4.1 for comparison with TRIC, using the same training set as in the previous experiment. We used our pairwise geometrical information feature y for state space x in the DPL notation, and $u_t = q_{t+1} - q_t$ for control u_t, and a neural network with 60 sigmoids to learn π, a regression problem with least-squares loss. We made the following observations for this regression problem that:

- DPL with loss Equation (2.6) could not learn even the train data and the regression loss remained very high.
- DPL with loss Equation (2.5) could learn the train data well, but had poor performance on the test data from the previous experiment.

Table 5.2 shows the errors of π on the train and test sets, indicating that probably much more data is necessary to learn a well generalizing mapping on test data from world state x to motion command u_t.

Table 5.2: The predictive accuracy for the DPL policy mapping $\pi : q_t \mapsto u_t$ using 27 trajectories of length 40 steps each. We show absolute values in radians of the joint speeds $|u_{t[k]}|$, and the absolute errors for train and test set errors $|u_{t[k]} - \pi(q_{t[k]})|$, averaged over the 14 joint dimensions $k \in \{1, 14\}$ and over all situations.

	average joint angle value in radians
joint speeds $\lvert u_{t[k]} \rvert$	0.009 ± 0.001
train set errors $\lvert u_{t[k]} - \pi(q_{t[k]}) \rvert$	0.002 ± 0.001
test set errors $\lvert u_{t[k]} - \pi(q_{t[k]}) \rvert$	0.006 ± 0.001

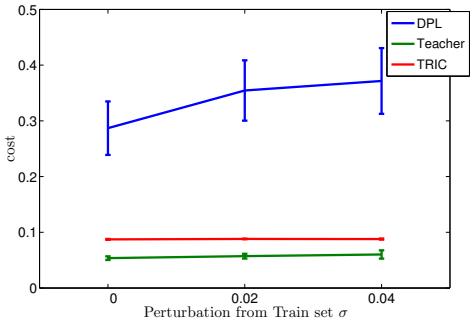


Figure 5.10: Comparison of TRIC to Direct Policy Learning on different test sets with increasing "distance" between training and test data of $\{0, 0.02, 0.04\}$ meters.

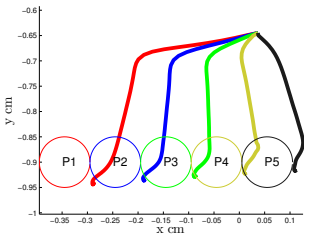
After we trained the policy π, we implemented an IK controller for DPL imitation using the following IK motion generation scheme:

$$C^{\text{DPL}}(q, q_t) = \|q - q_t\|^2 + 10^3 \|q - q_t - \pi(q_t)\|^2 + C_{\text{prior}}(q) \tag{5.17}$$

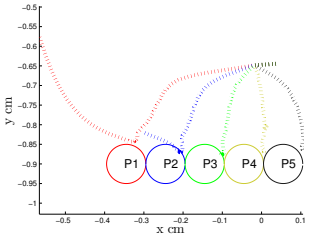
$$q_{t+1} = \underset{q}{\text{argmin}}\, C^{\text{DPL}}(q, q_t) \tag{5.18}$$

To test DPL performance more quantitatively we defined a new validation set, consisting of (a) 30 scenarios from the original train set, (b) 30 scenarios modified from the train set with added random translation to the target position sampled from a Gaussian with 2cm standard deviation, and (c) 30 scenarios created analogously with 4cm standard deviation. Figure 5.10 shows how the results degrade for DPL as the validation scenarios become remote from the training samples, whereas TRIC remains robust.

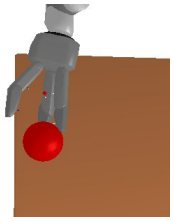
Figure 5.11(a) visualizes the motions generated by TRIC on 5 different target grasp object positions. For comparison, Figure 5.11(b) shows the trajectories of DPL for the same 5 targets. The 3 right-most positions (one of which is P3) are in the train set and DPL can put the fingers around the object. However, the fingers do not embrace the object as deeply and do not oppose each other as fully as expected, , see Figure 5.11(e). Thus the DPL grasp is worse than the teacher demonstration even in a situation from the train set (already seen target position). A reason can be that the collision term $g_{\text{collision}}(q)$ included in $C_{\text{prior}}(q)$ tries to push the hand away from the obstacle. Simply repeating old



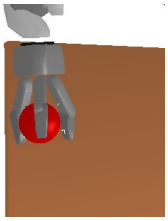
(a) TRIC grasps all 5 objects successfully. The controller converges to a good grasp posture.



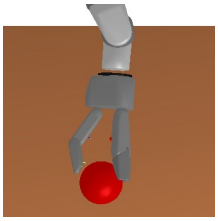
(b) DPL fails to grasp objects P1 and P2, and is suboptimal for objects P3, P4 and P5.



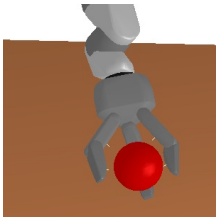
(c) At target location P1 method DPL can not grasp the object.



(d) Target P1 is grasped by TRIC.



(e) At target location P3 method DPL grasps suboptimally.



(f) Target P3 is grasped by TRIC.

Figure 5.11: A test to compare TRIC and DPL: 5 target sphere positions are to be grasped. Figures 5.11(a) and 5.11(b) show the trajectories of the robot finger landmark a_2 when grasping a sphere. The target (shown as a circle colored as the respective grasp trajectory) is translated on 5 positions.

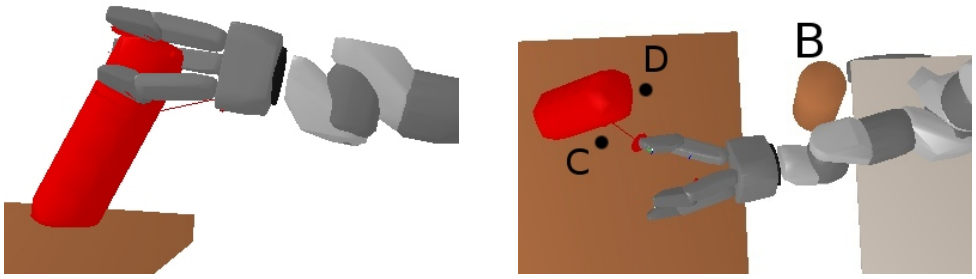
motion commands can not compensate adequately for this disturbance without properly tuned weights for the trade-off between collision and other motor commands.

In the 2 left-most positions (one of which is P1) DPL fails to make even a suboptimal grasp. These positions are not in the train set and this explain why DPL failed, see Figure 5.11(c). On all 5 positions TRIC manages to get good grasps comparable in quality to the teacher, see the examples of Figure 5.11(d) and 5.11(f) for grasp targets P1 and P3.

The relatively poor performance of DPL with joint space Q for the motion commands is to be expected, because as we reasoned already in Section 3.7, commands in joint space have difficulties in generalizing to translations in objects in the workspace. If the task is simply point reaching, this could be remedied by selecting a task space like $Y_{\text{target}} \in \mathbb{R}^3$, a low dimensional space that captures efficiently the essence of point reaching (see also Section 3.7). For grasping there are no such obvious solutions. Taking $\phi(q_{t+1}) - \phi(q_t)$ as motion command, in the high dimensional space of $y \in \mathbb{R}^{150}$ is not practical and worked even worse, because the machine learning prediction task becomes too complex - a mapping of dimensionality $\mathbb{R}^{150} \mapsto \mathbb{R}^{150}$.

Experiment 3 - Cylindrical Objects

In a third experiment we tested TRIC on more complex objects – an elongated cylinder with 30cm height and 5cm radius. We created a new training set of $N = 10$ demonstrations, by rotating the cylinder on its Y-axis randomly by $[0, \pi/2]$ radians.We then trained TRIC with the default parameters and tested the learned f on grasping of cylinders rotated differently than the train set. As can be seen in Table 5.3 the performance of TRIC with nonlinear model $K = 30$ is similar to Teacher. Linear TRIC ($K = 0$) was worse than the nonlinear one, thus, more complex motion policies require non-linear models for f. It is also worth noting that as few as $N = 10$ trajectories of length $T = 40$ were enough to train TRIC – efficiency in terms of demonstrations required. See Figure 5.12(a) for an example of a cylinder grasp performed by TRIC.



(a) A robot grasping pose for a cylinder.

(b) Grasping the target while avoiding obstacle B. The dots C and D indicate possible grasp approaches.

Figure 5.12: Implicit obstacle avoidance and cylinder grasping with TRIC.

Experiment 4 - Obstacle Avoidance as Motion Constraint

In a fourth experiment we examined how TRIC reacts to an obstacle B in its path to the target. Because our motion model includes a prior cost term $C_{\text{prior}}(q)$ for avoiding obstacles, the motion controller can handle such cases robustly. In our evaluation the obstacle B appears *only* in the validation set, not in the train set. TRIC is reasonably robust to these previously unseen obstacles: the motion rate controller tries to stay out

Table 5.3: Result for cylinder grasping on a set with $N = 10$ motions and default hyperparameters, TRIC with linear and nonlinear models compared with the teacher performance.

method	performance
Teacher	0.07 ± 0.01
TRIC $K = 30$	0.09 ± 0.02
TRIC $K = 0$	0.19 ± 0.02

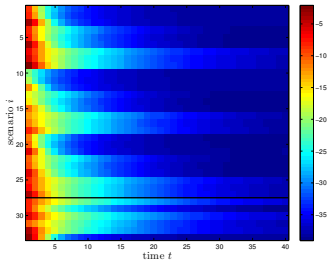


Figure 5.13: TRIC learns a value function $f(y)$ decreasing for all observed trajectories y_t^i for time steps $t \in \{1, ..., 40\}$ and situations $i \in \{1, ..., 33\}$. Situations 1 to 27 are from the train set, and situations 28 to 33 from the test set. Blue indicates low value.

of collisions and will adapt to a grasping pose that is compatible with this constraint, see Figure 5.12(b). The straight approach to grasp pose D is no longer feasible, because it will collide with B. Another grasping pose C can be found, one of the many grasp poses with low value f in the subspace of good grasps (see Figure 5.15). Because it is also reachable without collisions, C was preferred to D. This is an illustration of the generalization ability of TRIC unlike DPL, which is limited by the seen trajectories and can not adapt them to never before seen obstacles.

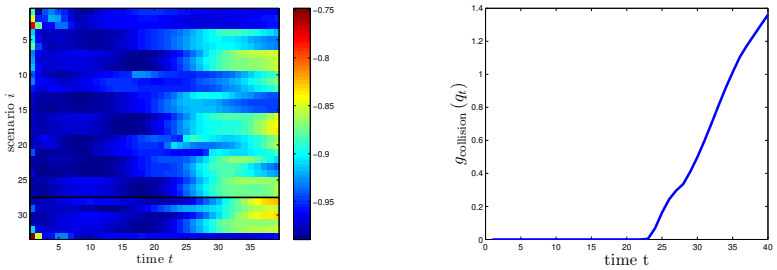
Clearly the local collision avoidance behavior of our controller has limitations – in a more complex world cluttered with obstacles reactive avoidance can get stuck in local minima, a drawback common to all local reactive controllers (see Section 2.3.3).

5.5.4 Analysis of the Trajectory Dataset: Structure Revealed from the Sparse Discriminative Value Function

Examining the Value Function f and its Gradient after Training

The TRIC models were trained robustly with respect to the different parameters. Figure 5.13 shows that we successfully learned a value function f always decreasing ahead in time, a learned invariant property of the demonstrated sphere grasping motions. We can also notice in this figure that we have identical behavior of this value function on the 27 training situations and the 6 test situations. This is an indication on how well our model generalizes from the train data to unseen novel situations in the same domain.

In Figure 5.14(a) we illustrate how well TRIC could align the gradient $\mathcal{J}$ to the trajectory relative steps $q_{t+1} - q_t$. In the majority of data points we had a value of $L_g(q_t^i)$ near -1, indicating that TRIC successfully learned a value function with this property. However, near the final time steps $t > 30$ this value worsens a bit. We have a possible explanation for this effect. Near the final phases of the motion the collision cost $g_{\text{collision}}$ (a term of C_{prior}) increases, as shown in Figure 5.14(b). This increase is due to the fact that the robot fingers close on the object surface near the end of a good grasp. The collision costs increase also the magnitude of the collision gradients. As a result, the value function is not decreased in the steepest descent manner, but in a way combined with



(a) Angle cosines for all situations and trajectory steps. Blue value indicates small cosine as desired in the loss criteria.

(b) A plot of the collision costs of a grasping motion: near the end the collisions rise significantly.

Figure 5.14: The value $L_g(q_t^i)$ reflecting cosine of angle between $\mathcal{J}$ and $q_{t+1} - q_t$ minimized by loss term L_g for all observed trajectories y_t^i for time steps $t \in \{1, ..., 40\}$ and situations $i \in \{1, ..., 33\}$. Situations 1 to 27 are in the train set, 28 to 33 in the test set.

collision avoidance, resulting in a less-steep value function decrease.

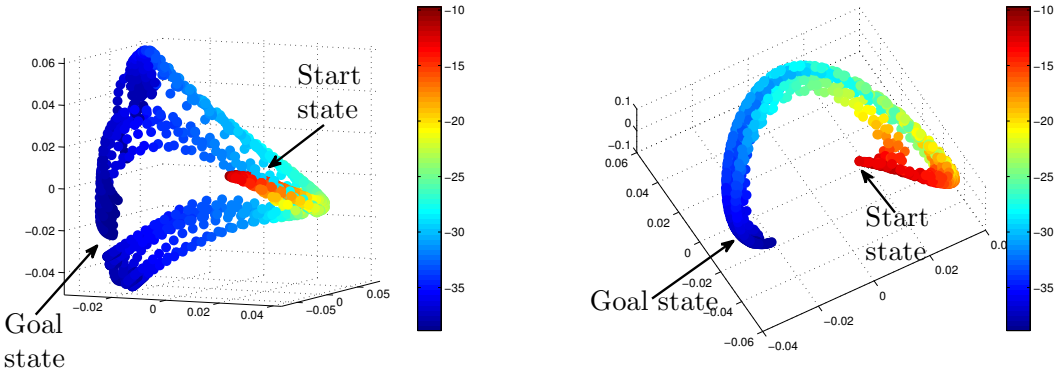
The Subspace of Good Grasps

Another way to analyze the effect of training TRIC is to look at the hidden layer $z = \frac{1}{1+e^{W_1 y}}$ of the neural network model for value f as defined in Equation (5.15), see Montavon et al. (2011). Figure 5.15 shows a low dim. kernel PCA visualization using Gaussian kernel on the raw features y and the learned hidden (latent) representations z of the 27 train trajectories.[3] As we showed in Figure 5.13, the trained value function decreases monotonically in time, so the color is also an indicator of the time progress of the sample trajectories, red is near $t = 1$ and $t = T = 40$ is dark blue. In Figure 5.15(a) the set of goal states ($t \approx T = 40$, colored blue) are scattered in a region and seem different depending on the different grasp posture. With such representation it is not easy to define what is a good grasp. Figure 5.15(b) shows that the training of TRIC produced such latent z values that all goal states are close to one another, much closer than in the first plot. Thus the trained neural network value function has found a compact representation of the subspace of good grasps. This representation allows TRIC to switch between different grasps when required, as in the experiment in Section 5.5.3.

5.5.5 Analysis of the Motion Features: Retrieving Relevant Task Spaces

We just showed the quality of the controller and value function learned by TRIC by examining the way it grasps. Another aspect of TRIC is the contribution of different features to the learned value function, which can lead to a better explanation of what exactly the robot learned from the teacher.

[3] Similar to the use of kernel PCA for visualization we showed in Section 3.7.1.



(a) Kernel PCA using Gaussian kernel on raw features y. Large goal state region.

(b) Kernel PCA using Gaussian kernel on hidden neural network layer z. Compact goal state representation.

Figure 5.15: Kernel PCA visualization of the trajectory data. The color indicates the value function f, which is inversely correlated with the time index t after proper training.

Sparsity of Features Contributing to the Value Function

To get an insight into the feature selection done implicitly by the loss minimization of Equation (5.4), we defined a score for the contribution of each feature to the value function f. It is defined as a weighted sum of the absolute values of coefficients of the model of f from Equation (5.16):

$$s(y) = |W_1^T||W_2| + |W_3| \tag{5.19}$$

This is a heuristic to see how much each feature contributes to the value function f. In Figure 5.16 we display these scores, using one of the trained models for the sphere grasping task. It can be seen that the L_1-regularization selected about 38 of these features for the cost f.

In order to interpret meaningfully the selected features we look at the geometric information of the landmarks (see Figure 5.8) represented by the features. Figure 5.17 shows the feature scores as a map related to pairwise measurements $p_j = (p_j^x, p_j^y, p_j^z)$ and norms $d_j = \|p_j\|$, as defined in Section 5.5.2. It can be seen that the highest score is $s(d_{1,2}) = 47.5$, which corresponds to the distance between the target object a_1 and one of the fingers a_2. The next best is $s(d_{1,4}) = 47.25$, distance between the target and another finger a_4. Overall, all 4 feature score maps in Figure 5.17 show that the features measuring relative distance of the target landmark a_1 from the fingers (column or row index 1) are the most important for the grasping task. This seems like a meaningful way to construct features for a grasping task.

The other geometric feature pairs with high scores, not involving the target object landmark a_1, are relative distances of the fingers with respect to one another and the base of the hand. These features influence the robot pose and the way the fingers align to

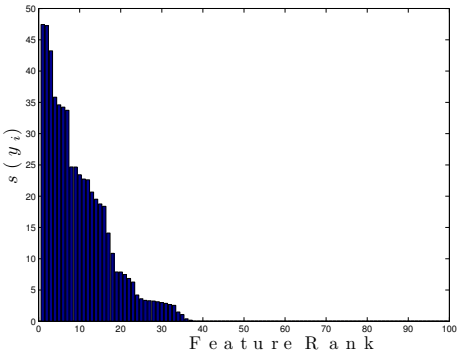


Figure 5.16: Effect of L_1-regularization: the motion features sorted by score $s(y_i)$ from the coefficients of the TRIC model. In our notation $s(y_i)$ is the score of feature i.

each other to get a grasp where the fingers oppose one another. This might seem obvious to the human observer who knows quite well what is grasping, but it is interesting to see what is the robot's estimation of the features required for imitation of grasping.

Analysis of the Gradients of the Value Function

Another interesting way to extract structure using TRIC is to look at the gradients $\frac{\partial f}{\partial y} \in \mathbb{R}^s$ evaluated at each data point y_t^i. This allows to analyze the time progress of the feature contribution to the activation of f. We can define the average gradient $\bigtriangledown f_t \in \mathbb{R}^s$ for a time step t:

$$\bigtriangledown f_t = \frac{1}{N} \sum_{i=1}^{N} \frac{\partial f}{\partial y}(y_t^i) \tag{5.20}$$

We are averaging the values of all evaluated gradients for a fixed time step. If we look at the individual dimensions of the vector $\bigtriangledown f_t$ this gives us the average contribution for time t of feature j to the value function.

Figure 5.18 illustrates the results, where we have displayed only the values of $\bigtriangledown f_t$ only for the most representative 20 dimensions by value of $s(y)$. We see how the contribution of each feature changes over time, and this can be interpreted as adaptable policy depending on the current state of the motions q_t. It is due to the nonlinear model we used for f with $K = 30$ sigmoid nodes. We see that in different time steps different features have the greatest contribution to f. For more complex motions with different phases we are likely to see even more variation of the gradients in time. A linear model (with $K = 0$ nonlinear sigmoid nodes) for f in Equation (5.16) would have constant gradients for every time step, equal exactly to W_3.

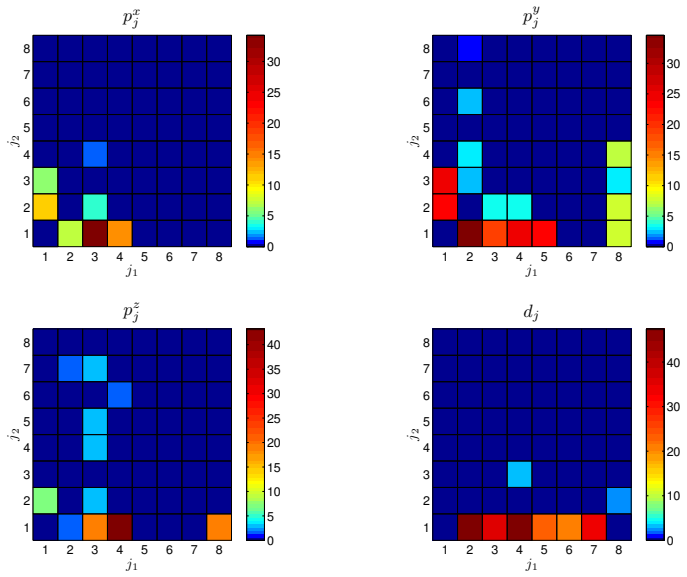


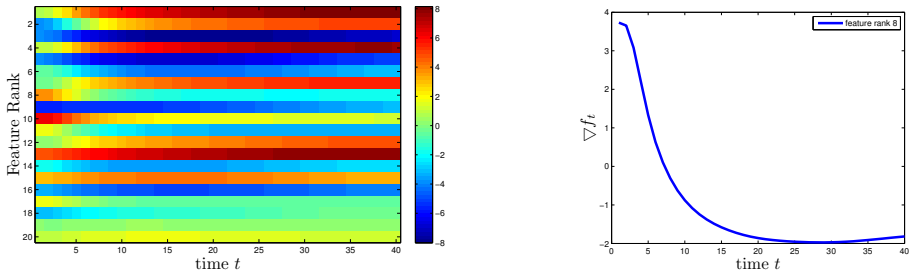
Figure 5.17: Four colormaps of the feature scores $s(y)$ represented as pairwise distances, red is high score and blue is low score. Each of the 4 maps corresponds to one of the three spatial dimensions $p_j = (p_j^x, p_j^y, p_j^z)$ or their norm $d_j = \|p_j\|$. Each entry of a map is a feature dimension indexed by $j = (j_1, j_2)$ and since we used 8 landmarks $j_1, j_2 \in \{1, 8\}$. In each map the column corresponds to j_1 and the rows corresponds to j_2, indicating the two compared landmarks.

The detailed view in Figure 5.18(b) shows $\bigtriangledown f_t$ just for the 8th best feature, which has the geometric interpretation of $p_{3,1}^y$, the y distance between the finger a_3 and the target a_1. We see that the gradient starts positive in the initial steps, indicating that the controller is decreasing the value of $p_{3,1}^y$ (thus bringing the finger closer to the target) in order to decrease $f(y)$. As time progresses the sign of $\bigtriangledown f_t$ changes and now this value $p_{3,1}^y$ will be increasing. This behavior has the following interpretation for the grasping task:

- First the robot hand approaches the sphere target, so the distance $p_{3,1}^y$ should get smaller.

- After the fingers are sufficiently close to the sphere surface, the distance $p_{3,1}^y$ should stay relatively constant; by having a value function that tries to increase $p_{3,1}^y$ we keep a balance between this and other geometrical features and the net effect is constant distance of fingers to target surface.

Note that the motions are not aligned perfectly in time, so this adds some distortion since $\bigtriangledown f_t$ averages for fixed time steps. A better analysis can be achieved by applying some form of Dynamic Time Warping for alignment in time (Myers and Rabiner 1981).



(a) The values of $\bigtriangledown f_t$ corresponding to the top 20 ranked features vs time t.

(b) Plot of the 8th feature by rank: the gradient changes sign at $t = 8$.

Figure 5.18: Average gradients $\bigtriangledown f_t$ of the value function f. Positive and negative values indicate how the respective pairwise object geometrical relation influences the f.

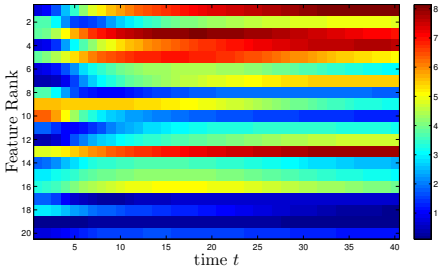


Figure 5.19: Average absolute gradients $\bigtriangledown f'_t$ of the value function f. The magnitude indicates how important a feature was on the average for time step t. Top ranked 20 features vs time t.

We can similarly define the average absolute gradient $\bigtriangledown f'_t \in \mathbb{R}^s$ for a time step t:

$$\bigtriangledown f'_t = \frac{1}{N} \sum_{i=1}^{N} |\frac{\partial f}{\partial y}(y_t^i)| \tag{5.21}$$

Figure 5.19 shows the values of $\bigtriangledown f'_t$. When time progresses features change their relative weight in the calculation of the value function. Some features are important at the beginning of the motion, when the robot arm is very far away from the target, other features are most important when the robot already has his fingers in a grasp on the object.

5.6 Conclusions from TRIC

In this chapter we presented a novel method for learning a sparse discriminative value function from demonstrations, finding relevant task spaces from a set of rich geometrical features, and generating motion with a controller using the value function. TRIC can

generalize robustly to different situations unseen during training. We tested its performance on a robot grasping task, presented results showing the effect of different training and model settings, and visualized the extracted task space features. We can use the learned model to interpret the most important task dimensions, and their relative weighting in time. A visualization of the hidden layers of the neural network we used for training reveals how TRIC represents a subspace of good grasping poses.

Our method has some limitations. First, it is not suitable for periodic movements and would need to add state information from previous time slices to deal with more complex motions. Second, we assume that an accurate robot kinematic model and sensors for workspace objects' locations are available that provide rich features. TRIC can be sensitive to noise in these input features and their Jacobians, and sometimes it is not possible or practical for the teacher to provide so much detailed motion information.

5.6.1 Future Work

A possible future research direction can explore even richer features and models to model cyclical behavior and movements with distinct subgoals and phases. By modifying the value function f to be a recursive or deep neural network, we will be able to model even more complex behaviors from demonstrations.

We can try to couple TRIC with powerful object representations like implicit surfaces.[4] Modeling more complex object geometries is important for realistic applications. Additionally, a coupling of perception and manipulation can be interesting in a framework where both objects and motions are represented as potential fields.

It is also possible to use TRIC on the huge amounts of recorded human motion data recently available online, e.g. the CMU Motion Capture Database (`mocap.cs.cmu.edu`). The analysis of human demonstrations with TRIC can both teach robots new skills and also uncover interesting results concerning which features underly human motion generation. Giving an accurate motion description – e.g. "use the elbow and never the wrist for the tennis forehand" – can benefit even humans wanting to learn a skill better.

We assumed that the data presented to TRIC comes from a single task. It can be an interesting challenge to use trajectories that come from different tasks (e.g. grasping as one task and throwing as a second) where the task identity is hidden. We can use value function models with a latent variable for the hidden task identity to simultaneously group the different motions into their respective tasks and learn the value function.

So far we have used TRIC in settings where no task cost information is known. We can try setups where such information is available (at least partially) and directly incorporate cost observations into the training loss.

[4] Implicit surface object models are learned from sensory data and the object surface is a nonlinear function potential itself, e.g. a Gaussian Process or SVR, see Steinke et al. (2005).

Chapter 6

Conclusions

In this thesis we showed how a novel fusion of machine learning and robotic methods can be of great benefit for creating and understanding robot motions. Our contributions can be grouped in two main categories. First, our algorithms TP and TRIC constitute fast motion generation methods capable of executing complex tasks. Second, we presented new methods for analysis of the feature representations that build the robot's inner view of the tasks and workspace situations it has to deal with. Our methods were defined in a relatively general way, and can be adapted to various worlds, tasks and robots. For example. the basic features we used for situations and motions can be augmented with additional geometric information and transformations in order to represent even richer structures in data. The machine learning methods we used for situation classification (TP) and value function approximation (TRIC) can be enhanced using various algorithms from the rich toolbox of machine learning. Using special kernels or neural networks with different architecture can lead to effective adaptations of our methods for various datasets. We believe that this fusion of data intensive techniques and robot algorithms can be of great benefit for any motion generation problem satisfying the relatively general assumptions we made. This assumptions are that the world situations have a certain structure and can be thus mapped to different motions, and that a large amount of information is available, providing features both for describing world structure and robot motion task spaces.

6.1 What is Possible Now: Summary of the Benefits of TP and TRIC

In order to show the practical utility of our methods we will use the example of an industrial factory with robot arm manipulators soldering car metal frames (targets), as shown on Figure 6.1. Such a setup is suitable for gathering data and using machine learning: a task is to be repeated multiple times in some structured domain. Even if the workspace is not static – target objects move on a conveyor belt, other machines and humans get in the way – we can expect that experience of the environment and

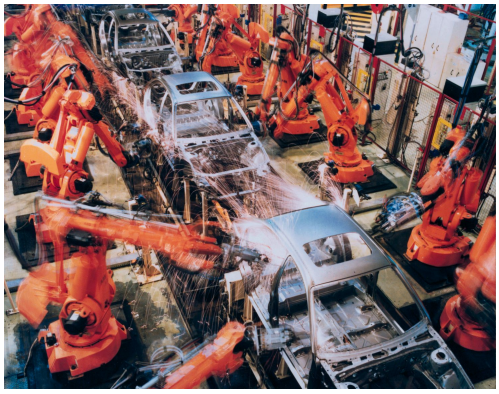


Figure 6.1: Factory setting for robot motion and manipulation tasks: multiple robot arms (model ABB IRB 6400) have to solder an object. Image taken from the producer's website `www.abb.com`.

successful motions can improve the motion generation process.

The most basic approach would be to have a fixed motion trajectory that the machines are programmed to follow, but this has great limitations. First, generating a fixed trajectory just by expert knowledge can be quite involved in itself. Second, such a trajectory would work well just for a fixed position of the robot and its manipulation target. It is more realistic to expect that something is different every time the robot arm approaches its soldering target:

- the manipulation target is not where the robot expects it to be, e.g. due to issues with time lag and other problems with the conveyor belt
- a human or other robot stands in the way and a fast generation of a new trajectory with good collision avoidance is required to avoid accidents
- the target objects are of different geometries, and a variety of motions will be required for the same task, making it impossible to enumerate all of them

It is possible that an expert can specify the desired positions of the endeffector with respect to the target object that result in a good soldering manipulation. Such specification, together with priors to avoid collisions and other dangerous motions that can break the robot (e.g. joint limits), can be used to define a cost function for the manipulation task. Then we can use planning algorithms to calculate optimal trajectories for the task in simulation and then execute them for real. Standard motion planning algorithms can be used to create motions of high quality, often much better than what a human can program. If we want to further speed-up and improve planning, our method TP from Chapter 3 can be of good use. In order to use TP to predict motion trajectories we need to do these steps:

- design a situation descriptor

  - either define a set of important landmarks in the workspace and take all pairwise distances between them, or use the output of a sensor (e.g. a camera) as a descriptor of space around the robot
  - if the objects to manipulate are moving, include velocity information
  - if the objects are of varying sizes, add some size information as features
- make a database by recording situation descriptors and optimal trajectories calculated by a motion planner using the cost function we assumed to have
- to reuse the database motions specify a reasonable task space for transfer, e.g. the relative positions of robot endeffector in the frame of the target object to manipulate

We would get the following benefits after we train TP:

- a potentially great speed-up of motion planning convergence speed, allowing the robot to generate better motions in less time
- an analysis of the datasets of workspace situations and optimal trajectories, revealing situation features that are important for the task execution and influence the type of optimal motions

If no cost or other task specification can be designed, we can take the programming by demonstration approach in order to design flexible controllers for robot motion generation. This is often applicable to scenarios where the human experts in the factory know what is a good manipulation by their experience. and can do it in some way, but can not formalize a good manipulation in the form of a function with specific motion features as inputs. In order to use programming by demonstration with TRIC (Chapter 5), we would need to provide enough data that can be used as input to our algorithm :

- define rich task spaces that incorporate multiple geometric features of the robot and additional information from relevant target objects in the world
- generate a set of good example motions using human input (e.g. servoing or teleoperating the robot)
- record these motions in the rich task spaces

Training TRIC and using it to learn the structure of the recorded motions can give the following benefits:

- a controller that can execute the inferred motions and allow the robot arm to generalize the desired manipulation task to changing world conditions
- the above controller can be also fused with priors on good motions like collision and joint limit avoidance
- a value function that indicates the goals of the controller
- selection of the features important for the task, giving an intuitive geometric description of the structure of motions good for the task

The analysis of the learned value function and motion features using the approaches we showed in Section 5.5.4 and Section 5.5.5 can also provide useful feedback to humans wanting to be sure that the robot has learned a useful motion model. If necessary, the motion data used as input can be changed and the rich task spaces defined for the task can be augmented with other potentially important manipulation features.

6.2 Future Work: Fusing TP and TRIC

The two methods we presented, TP for planning and TRIC for imitation learning, have been developed with different applications in mind. A future combination of the two seems very promising because the two approaches can augment each others' strengths. We will sketch some potential ways to make this combination:

- **TRIC into TP** – we can use TRIC to analyze the datasets of optimal motions available in the TP setting. With TRIC we can find out which task spaces are best suited to repeat and generalize demonstrated motions. This would be a more principled way than the task space pooling approach[1] we used so far in TP to find task spaces that reflect the structure of the stored trajectories and can improve planning results significantly. Once such optimal task spaces are found, we can either use them just to represent the recorded trajectories in such task spaces and use cost sensitive classification for prediction, or leave the trajectory transfer components and directly use TRIC as replacement of the Transfer IK initialization operator.
- **TP and planning into TRIC** – a combination of local planners (like iLQG) with TRIC can also be beneficial. The straightforward way to proceed can be described in three steps. First, learn a value function with TRIC as described already. Second, add the value function as a term in a specially defined trajectory cost function (with additional smoothness and collision terms) requiring to go to a state of minimal value at the final time step T. Third, use a local planner to find a trajectory of fixed time duration minimizing this cost function. This could be a more efficient way than the reactive motion rate control to combine motion priors like obstacle avoidance with the decreasing value function criteria. The planner will ensure that the value function decreases smoothly and the robot moves gradually to its target, going to a region where the value function is low (and the task is achieved) and where additional constraints like collision avoidance are satisfied. Such a planner can probably deal better than pure IK control with degenerate value surfaces where no constant descent is possible. We tested a prototype implementation of such a combination of TRIC with a local planner, and got promising first results.

[1] A few candidate task spaces were directly tested for motion planning and the best were selected.

Appendix A

Proofs and Derivations

A.1 Derivation of the Gradient of the Training Loss for TP with NNOpt

We will now write formally the partial derivative of the loss of TP as defined in Equation (3.22), with respect to the vector w.

The final derivative of the loss is:

$$\frac{\partial L(w; D')}{\partial w} = \frac{1}{|D^x|} \sum_{x \in D^x} \frac{\partial \mathrm{E}\left\{F(x, f(x))\right\}}{\partial w} + \lambda \operatorname{sign}(w) \tag{A.1}$$

Here the vector of the ± 1 signs of w is defined as gradient of the L_1 norm, see Schmidt et al. (2007).

The derivative of the expectation Equation (3.21) is:

$$\frac{\partial \mathrm{E}\left\{F(x, f(x))\right\}}{\partial w} = \sum_{i=1}^{d} F(x, \mathcal{T}_{xx_i} \boldsymbol{q}_i) \frac{\partial P(f(x) = \mathcal{T}_{xx_i} \boldsymbol{q}_i)}{\partial w} \tag{A.2}$$

The derivative of the full probability $P(f(x) = \mathcal{T}_{xx_i} \boldsymbol{q}_i)$ is:

$$\frac{\partial P(f(x) = \mathcal{T}_{xx_i} \boldsymbol{q}_i)}{\partial w} = \frac{Z \frac{\partial k(x, x_i)}{\partial w} - P(f(x) = \mathcal{T}_{xx_i} \boldsymbol{q}_i) \sum_{j=1}^{d} \frac{\partial k(x, x_j)}{\partial w}}{Z^2} \tag{A.3}$$

The derivative of $k(x, x_i)$ is:

$$\frac{\partial k(x, x_i)}{\partial w} = \frac{\partial \exp\{-\frac{1}{2}(x - x_i)^T W (x - x_i)\}}{\partial w} = -k(x, x_i)(x - x_i)^2 \operatorname{diag}(w) \tag{A.4}$$

A.2 Proof of Proposition 3.5.2 for SVR complexity

Proposition: The algorithmic complexity for training the SVR predictor from Equation (3.24) is $O\left(d(d_x^\alpha + s d_x^2)\right)$ with $2 < \alpha < 3$.

Proof The calculation of the polynomial kernel matrix takes time $O(sd_x^2)$ for d_x points in $\mathbb{R}^s$. The training time of the SVR is stated in the literature (Bordes et al. 2005) as $O(d_x^\alpha)$ with $2 < \alpha < 3$. This should be multiplied with d, because we train d multiple regression models. Thus, the final complexity $O\left(d(d_x^\alpha + sd_x^2)\right)$ is proven.

A.3 Proof of Proposition 3.5.1 for NN complexity

Proposition The algorithmic complexity of minimizing loss 3.22 with a second order optimizer (BFGS) is $O(d_x d_s + s^2)$.

Proof We have s dimensions of each situation descriptor. The complexity to calculate prediction probabilities for a single situation is $O(ds)$ because we calculate similarities to d other trajectories, and each similarity evaluation $k(x_i, x_j)$ takes s operations. For d_x situations the complexity is $O(d_x ds)$ to evaluate all prediction probabilities and calculate the gradient of the loss. Once we have calculated the loss value and gradient we make a gradient descent step using the BFGS second order method for unconstrained gradient-based optimization, see Nocedal and Wright (2006). It requires usually a small amount of iterations to converge. The costs per iteration are of complexity $O(s^2)$ due to the approximation of Hessian matrix. Combining the costs of calculation of the gradient and estimating the next BFGS step results in complexity $O(d_x ds + s^2)$.

A.4 Proof of Proposition 5.2.1 for the Direction of IK Generated Motion Steps

Proposition If $\delta_1 \to \infty$ then the IK solution q_{t+1} minimizing Equation (5.2) has the property that the next step $q_{t+1} - q_t$ is proportional to the value function gradient $\mathcal{J}$.

Proof If $\delta_1 \to \infty$ then the term $\|f \circ \phi(q) - f \circ \phi(q_t) + \delta_2\|_{\delta_1}^2$ of Equation (5.2) is weighted so high that C_{prior} and any other cost terms we might add are neglected. Let $\mathcal{J}$ be the gradient of the value function $f \circ \phi(q)$ evaluated at $q = q_t$. Using the linearization $f \circ \phi(q_{t+1}) = f \circ \phi(q_t) + \mathcal{J}(q_{t+1} - q_t)$, we can apply the IK Equation (2.1):

$$q_{t+1} = q_t - \delta_2 \mathcal{J}^\sharp \tag{A.5}$$

$$\mathcal{J}^\sharp = \left(\mathcal{J}^T \delta_1 \mathcal{J}\right)^{-1} \mathcal{J}^T \delta_1 \tag{A.6}$$

$$= \mathcal{J}^T \left(\mathcal{J}\mathcal{J}^T + (\delta_1)^{-1}\right)^{-1} = \frac{1}{\|\mathcal{J}\|^2} \mathcal{J}^T \tag{A.7}$$

We have used the Woodbury identity and the fact that $\mathcal{J}\mathcal{J}^T = \|\mathcal{J}\|^2$ in the case where we have a 1-dimensional task variable y (in that case the Jacobian is a row vector gradient). $\mathcal{J}^\sharp$ is called the pseudoinverse of $\mathcal{J}$. Thus, the steps generated by our motion model are proportional to $\mathcal{J}$ times a negative scalar number.

A.5 TRIC Complexity: Training Loss and Motion model

Let us call O_f the complexity of evaluating the value function $f \circ \phi$. We also write O_{df} for the complexity of calculating the first derivative and O_{ddf} for the complexity of evaluating the second derivative of f. The complexities of O_f, O_{df} and O_{ddf} are at least linear in s, where s is the number of dimensions of $y = \phi(q)$, but this is highly specific for the choice of value function model f, see for example Section 5.5.2 for a concrete example. Extracting sparse features can speed-up the calculation of task spaces $\phi(q)$ and their derivatives $\frac{\partial \phi}{\partial q}$, and decrease the complexities O_f and O_{df}, if we use this sparsity in the implementation.

The complexity of L_n is $O_n = O(M(O_{df} + O_f))$ because for each noisy sample we have to calculate its value function and the gradient. The complexity of L_g is $O_g = O(O_{ddf} + O_{df})$, because we have to optimize the Jacobian instead of the value function directly. The total loss function complexity is then $O(NT(O_n+O_g))$, because we calculate L_g and L_n for each step of every trajectory.

Making an IK step with joints $q \in \mathbb{R}^n$ is of complexity $O(O_f O_{df} + n^3)$ because we need to invert a matrix of size $n \times n$ and calculate the value and gradient of the value function, see Equation (2.1).

A.6 Proof of Lemma 5.3.1 for the TRIC Value Function

Lemma Assume that the following holds for TRIC and the training demonstrations:
(i) a single linear trajectory $\{q_t\}_{t=1}^T$ in the joint task space $\phi(q) = q \in \mathbb{R}^n$,
(ii) $M \to \infty$ noisy samples,
(iii) $\text{sign}(a-b)$ is used instead of $\log(1+e^{a-b})$ in the terms of L_n in Equation (5.9),
(iv) values of the metric d_{curve} in Equation (5.7) smaller than a constant ϵ_0 are set to 0.
Then a lower bound L_B on the loss $L' = \sum_{t=1}^T L_n(q_t; w)$ exists, and when $L' = L_B$ it holds for all time steps t that (I) $f(q_{t-1}) > f(q_t)$, i.e. the value function is decreasing along the trajectory, and (II) $f(q_T)$ is a minimum of the value function.

Proof

- Using (i) we assume the trajectory starts at the origin of some coordinate frame and is aligned with an axis: $q_1 = (0, 0, ...)$ and $q_T = (Q, 0, ...)$ where Q is the length of the trajectory in joint space.
- Let $\zeta_t = \{\tilde{q}_{t,m}\}_{m=1}^M$ be the set of noisy samples for t, sampled as in Equation (5.5).
- By (iii) we write the loss as $L' = \sum_{t=1}^T \sum_{\tilde{q}_{t,m} \in \zeta_t} \epsilon_{t,m} \text{sign}(f(q_t) - f(\tilde{q}_{t,m}))$. We state

that the following property (A) holds:

$$\left[L' = -\sum_{t=1}^{T}\sum_{m\in\zeta_t}\epsilon_{t,m} = L_B\right] \Leftrightarrow \left[\forall t, m : \Big[\epsilon_{t,m} > 0 \Rightarrow f(\tilde{q}_{t,m}) > f(q_t)\Big]\right] \quad \text{(A.8)}$$

We will now prove (A):
– in the $\Leftarrow$ direction: each $\epsilon_{t,m} \geq 0$ by Equation (5.8) and $\text{sign}(a) \in \{-1, 0, 1\}$, thus it is clear that the lower bound is achieved if $\forall t, m : \Big[\epsilon_{t,m} > 0 \Rightarrow f(\tilde{q}_{t,m}) > f(q_t)\Big]$.
– in the $\Rightarrow$ direction: suppose some $\epsilon_{t,k} > 0$ and $\text{sign}(f(q_t) - f(\tilde{q}_{t,k})) \geq 0$; then another value function f' identical to f everywhere but in $\tilde{q}_{t,k}$ – i.e. $\text{sign}(f'(q_t) - f'(\tilde{q}_{t,k})) < 0$ – will have a lower value and so the lower bound was not achieved by the first f.

- We will show that (A) implies (I) and (II). Because of (ii) each ζ_t can be assumed to fill densely $\mathbb{R}^n$, i.e. there is always a sample arbitrarily close to any point in space. Thus (ii) implies that $\forall t \exists \tilde{q}' \in \zeta_t$ s.t. $\tilde{q}'$ is arbitrarily close to q_{t-1}. Such a sample will have weight $\epsilon' > 0$ by Equation (5.8). Thus (A) implies that $f(q_{t-1}) > f(q_t)$ for all t, and (I) is valid.

- On the other side, $\forall t \exists \tilde{q}'' \in \zeta_t$ s.t. $\tilde{q}''$ is arbitrarily close to q_{t+1}. However, the corresponding weight $\epsilon'' = 0$ by Equation (5.8) and thus $f(q_{t+1}) > f(q_t)$ is not implied by (A).

- Using (ii) again, ζ_T contains a dense neighborhood around q_T, and all weights $\epsilon_{T,m} > 0$ by Equation (5.8). Thus by (A) $\forall \tilde{q}_{T,m} \in \zeta_T : f(q_T) < f(\tilde{q}_{T,m})$, and so $f(q_T)$ is a minimum, (II).

- Now we will show constructively that the lower bound L_B can be reached by a specific choice of value function f.
– Define $f(q) = (q_{[1]})^2 - 2Qq_{[1]} + \frac{Q^2+1}{\epsilon_0} d_{\text{curve}}(q, \{q_t\}_{t=1}^T)$, where the square brackets indicate dimension.
– The minimum of f is at $q = Q$, because $(q_{[1]})^2 - 2Qq_{[1]}$ is a quadratic form and $d_{\text{curve}} \geq 0$ by Equation (5.7). We note that $\max_{t\in\{1,T\}} f(q_t) = 0$, using that $d_{\text{curve}}(q, \{q_t\}_{t=1}^T) = 0$ for the demonstrated trajectory samples.
– Choose a random sample $\tilde{q}^* \in \zeta_t$ for any t, with weight ϵ^*. By Equation (5.8) and (iv) it holds $\epsilon^* \geq d_{\text{curve}}(\tilde{q}^*, \{q_t\}_{t=1}^T)$.
– If $d_{\text{curve}}(\tilde{q}^*, \{q_t\}_{t=1}^T) = 0$ then $\tilde{q}^*$ lies on the trajectory and $\tilde{q}^* = (\gamma, 0, 0, ...)$ for $\gamma \in [0, Q]$. If $\gamma < t$ by the construction of f it holds that $f(\tilde{q}^*) > f(q_t)$ and by Equation (5.8) $\epsilon^* > 0$. If $\gamma \geq t$ then by Equation (5.8) $\epsilon^* = 0$.
– If $d_{\text{curve}}(\tilde{q}^*, \{q_t\}_{t=1}^T) > \epsilon_0$ then $\epsilon^* > \epsilon_0$. By construction of f it holds $f(\tilde{q}^*) > 1 > f(q_t)$ because $f(q_t) \leq 0$.
– We examined all possible cases for $\tilde{q}$, and (A) always holds. Thus the lemma is proven.

By taking $\epsilon_0 \to 0$ we can effectively relax (iv). Assumption (i) can be relaxed and our proof can be generalized using some additional transformations to any curved path which is not crossing itself. In general it is not possible to prove the lemma without assumption (ii) and for multiple different trajectories: one can construct data for which the lemma does not hold. However, properties (I) and (II) seem to hold in general for various datasets we tried.

A.7 Proof of Proposition 5.3.4 for Lyapunov Attractor Properties of TRIC

Proposition Suppose we have trained TRIC on a single trajectory $\{q_t, y_t\}_{t=1}^{T}$ and that the implications of Lemma 5.3.1 hold for a trajectory in that task space. Additionally, we generate motion with $\delta_1 \to \infty$, i.e. very high weighting of the value function. Then the motion generated by the model in Equation (5.1) fulfills the conditions of Theorem 5.3.3 and is thus asymptotically stable in the joint space subspace $Q' = \{q' : \phi(q') = \phi(q_T) = y_T\}$.

Proof Because of $\delta_1 \to \infty$ the term decreasing the value function f will dominate the motion equation and we can ignore the effect of the other terms. Let's construct $V(q) = f \circ \phi(q) - c_T$, where $c_T = f \circ \phi(q_T)$ is the value function evaluated at joint state q_T (last trajectory step) after training of TRIC. Then the Lyapunov stability conditions hold:

- (a) holds because of Lemma 5.3.1 which implies that $c_T = f \circ \phi(q_T)$ is local minimum and any other joint state q s.t. $\phi(q) \neq \phi(q_T)$ will have higher value $f \circ \phi(q)$.
- (b) holds by the construction of $V(q)$ directly implying that

$$V(q_T) = f \circ \phi(q_T) - c_T = 0$$

- Proposition 5.2.1 holds because we assumed that $\delta_1 \to \infty$. This implies that the steps of the motion model are proportional to the gradient $\mathcal{J}$. Thus the motion model of Equation (5.2) will make steps decreasing the value $f \circ \phi(q_t)$ constantly and (c) holds.
- (d) holds because $c_T = f \circ \phi(q_T)$ is local minimum and after we reach a joint state q' s.t. $\phi(q') = \phi(q_T)$ the gradient of the value function will be 0 and no further decrease will be possible.

Literature

B. D. Argall, S. Chernova, M. M. Veloso, and B. Browning. A survey of robot learning from demonstration. *Robotics and Autonomous Systems*, 57(5):469 – 483, 2009.

C. G. Atkeson. Using local trajectory optimizers to speed up global optimization in dynamic programming. In *NIPS*, pages 663–670, 1993.

R. E. Bellman, editor. *Dynamic Programming*. Princeton University Press, 1957.

M. Berniker and K. Kording. Estimating the sources of motor errors for adaptation and generalization. *Nature Neuroscience*, 11(12):1454–1461, 2008.

A. Billard, Y. Epars, S. Calinon, G. Cheng, and S. Schaal. Discovering optimal imitation strategies. *Robotics and autonomous systems, Special Issue: Robot Learning from Demonstration*, 47(2-3):69–77, 2004.

A. Bordes, S. Ertekin, J. Weston, and L. Bottou. Fast kernel classifiers with online and active learning. *Journal of Machine Learning Research*, 6:1579–1619, 2005.

M. Branicky, R. Knepper, and J. Kuffner. Path and trajectory diversity: Theory and algorithms. In *IEEE Int. Conf. on Robotics and Automation (ICRA)*, pages 1359–1364, 2008.

M. L. Braun, J. M. Buhmann, and K.-R. Müller. On relevant dimensions in kernel feature spaces. *Journal of Machine Learning Research*, 9:1875–1908, 2008.

O. Brock and O. Khatib. Elastic strips: A framework for integrated planning and execution. In *The Sixth International Symposium on Experimental Robotics*, pages 329–338, 2000.

S. Calinon and A. Billard. Recognition and reproduction of gestures using a probabilistic framework combining PCA, ICA and HMM. In *22nd Int. Conf. on Machine Learning (ICML)*, pages 105–112, 2005.

S. Calinon and A. Billard. Incremental learning of gestures by imitation in a humanoid robot. In *HRI '07: Proceedings of the ACM/IEEE international conference on Human-robot interaction*, pages 255–262, 2007.

J. Call and M. Carpenter. Three sources of information in social learning. *Imitation in animals and artifacts*, pages 211–228, 2002.

S. Caselli and M. Reggiani. Randomized motion planning on parallel and distributed architectures. In *Euromicro Conf. on Parallel, Distributed, and Network-Based Processing*, page 297, 1999.

D. Challou, D. Boley, M. Gini, and V. Kumar. A parallel formulation of informed randomized search for robot motion planning problems. In *IEEE Int. Conf. on Robotics and Automation (ICRA)*, pages 709–714, 1995.

J. T. Coull, F. Vidal, C. Goulon, B. Nazarian, and C. Craig. Using time-to-contact information to assess potential collision modulates both visual and temporal prediction networks. *Frontiers in human neuroscience 2008;2:10*, 2008.

J. J. Craig. *Introduction to Robotics: Mechanics and Control*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1989. ISBN 0201095289.

I. Dinstein, C. Thomas, M. Behrmann, and D. J. Heeger. A mirror up to nature. *Current biology : CB*, 18(1), 2008.

S. Dragiev, M. Toussaint, and M. Gienger. Gaussian process implict surface for object estimation and grasping. In *IEEE Int. Conf. on Robotics and Automation (ICRA)*, 2011.

P. Dyer and S. R. McReynolds. *The Computation and Theory of Optimal Control*. Elsevier, 1970.

A. Elfes. Using occupancy grids for mobile robot perception and navigation. *Computer*, 22(6):46–57, 1989.

D. Ferguson, N. Kalra, and A. T. Stentz. Replanning with rrts. In *IEEE Int. Conf. on Robotics and Automation (ICRA)*, pages 1243 – 1248, May 2006.

P. F. Ferrari, V. Gallese, G. Rizzolatti, and L. Fogassi. Mirror neurons responding to the observation of ingestive and communicative mouth actions in the monkey ventral premotor cortex. *European Journal of Neuroscience*, 17(8):1703–1714, 2003.

M. Gienger, M. Toussaint, N. Jetchev, A. Bendig, and C. Goerick. Optimization of fluent approach and grasp motions. In *8th IEEE-RAS International Conference on Humanoid Robots*, 2008.

M. Haindl, P. Somol, D. Ververidis, and C. Kotropoulos. Feature selection based on mutual correlation. In *CIARP*, pages 569–577, 2006.

K. Hiraki, A. Sashima, and S. Phillips. From Egocentric to Allocentric Spatial Behavior: A Computational Model of Spatial Development. *Adaptive Behavior*, 6(3-4):371–391, 1998.

E. S. L. Ho, T. Komura, and C.-L. Tai. Spatial relationship preserving character motion adaptation. *ACM Transactions on Graphics*, 29(4):1–8, 2010.

M. Howard, S. Klanke, M. Gienger, C. Goerick, and S. Vijayakumar. Behaviour generation in humanoids by learning potential-based policies from constrained motion. *Applied Bionics and Biomechanics*, 5(4):195–211, 2008.

M. Howard, S. Klanke, M. Gienger, C. Goerick, and S. Vijayakumar. A novel method for learning policies from variable constraint data. *Autonomous Robots*, 27:105–121, 2009.

C. Igel, M. Toussaint, and W. Weishui. Rprop using the natural gradient. *Trends and Applications in Constructive Approximation. International Series of Numerical Mathematics*, 151:259–272, 2005.

W. Ilg, G. H. Bakir, J. Mezger, and M. A. Giese. On the representation, learning and transfer of spatio-temporal movement characteristics. *Int. Journal of Humanoid Robotics*, 1(4):613–636, 2004.

O. C. Jenkins and M. J. Matarić. A spatio-temporal extension to isomap nonlinear dimension reduction. In *21st Int. Conf. on Machine Learning (ICML)*, 2004.

N. Jetchev and M. Toussaint. Trajectory prediction: Learning to map situations to robot trajectories. In *26th Int. Conf. on Machine Learning (ICML)*, pages 449–456, 2009.

N. Jetchev and M. Toussaint. Trajectory prediction in cluttered voxel environments. In *Int. Conf. on Robotics and Automation (ICRA)*, pages 2523–2528, 2010.

N. Jetchev and M. Toussaint. Task space retrieval using inverse feedback control. In *28th Int. Conf. on Machine Learning (ICML)*, pages 449–456, 2011a.

N. Jetchev and M. Toussaint. Fast motion planning from experience: Trajectory prediction for speeding up movement generation, 2011b. Submitted.

L. E. Kavraki, J.-C. Latombe, R. Motwani, and P. Raghavan. Randomized query processing in robot path planning. In *Twenty-seventh annual ACM Symposium on Theory of Computing (STOC)*, pages 353–362, 1995.

S. M. Khansari-Zadeh and A. Billard. Bm: An iterative algorithm to learn stable non-linear dynamical systems with gaussian mixture models. In *IEEE Int. Conf. on Robotics and Automation (ICRA)*, pages 2381–2388, 2010.

G. Konidaris and A. Barto. Autonomous shaping: knowledge transfer in reinforcement learning. In *23rd Int. Conf. on Machine Learning (ICML)*, pages 489–496, 2006.

O. Kroemer, R. Detry, J. H. Piater, and J. Peters. Active learning using mean shift optimization for robot grasping. In *IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS)*, pages 2610–2615, 2009.

O. Kroemer, R. Detry, J. H. Piater, and J. Peters. Grasping with vision descriptors and motor primitives. In *ICINCO (2)*, pages 47–54, 2010.

M. Krutzen, J. Mann, M. Heithaus, R. Connor, L. Bejder, and W. Sherwin. Cultural transmission of tool use in bottlenose dolphins. *Proceedings of the National Academy of Sciences*, 102(25):8939–8943, 2005.

S. M. LaValle. *Planning Algorithms*. Cambridge University Press, Cambridge, U.K., 2006. Available at http://planning.cs.uiuc.edu/.

Y. LeCun, S. Chopra, R. Hadsell, M. Ranzato, and F. Huang. A tutorial on energy-based learning. In *Predicting Structured Data*, 2006.

M. Linderoth, A. Robertsson, K. Aström, and R. Johansson. Object tracking with measurements from single or multiple cameras. In *Int. Conf. on Robotics and Automation (ICRA)*, pages 4525–4530, 2010.

D. G. Lowe. Similarity metric learning for a variable-kernel classifier. *Neural Computation*, 7:72–85, 1995.

J. B. MacQueen. Some methods for classification and analysis of multivariate observations. In *Proc. of the 5th Berkeley Symposium on Mathematical Statistics and Probability*, volume 1, pages 281–297, 1967.

S. Martin, S. Wright, and J. Sheppard. Offline and online evolutionary bi-directional RRT algorithms for efficient re-planning in dynamic environments. In *IEEE Int. Conf. on Automation Science and Engineering (CASE).*, pages 1131–1136, 2007.

A. McGovern and R. S. Sutton. Macro-actions in reinforcement learning: an empirical analysis. Technical Report 98-70, University of Massachusetts, Amherst, 1998.

G. Montavon, M. Braun, and K.-R. Müller. Kernel analysis of deep networks. *Journal of Machine Learning Research*, 12:2563–2581, 2011.

M. Muehlig, M. Gienger, J. J. Steil, and C. Goerick. Automatic selection of task spaces for imitation learning. In *IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS)*, pages 4996–5002, 2009.

F. A. Mussa-Ivaldi and E. Bizzi. Motor learning through the combination of primitives. *Philosophical transactions of the Royal Society of London.*, 355:1755–69, 2000.

C. S. Myers and L. R. Rabiner. A comparative study of several dynamic time-warping algorithms for connected word recognition. *The Bell System Technical Journal*, 60(7): 1389–1409, 1981.

A. Nakhaei and F. Lamiraux. Motion planning for humanoid robots in environments modeled by vision. In *8th IEEE-RAS Int. Conf. on Humanoid Robots*, pages 197–204, 2008.

J. Nocedal and S. Wright. *Numerical optimization*. Springer series in operations research and financial engineering. Springer, New York, NY, 2nd edition, 2006.

K. Pearson. On lines and planes of closest fit to systems of points in space. *Philosophical Magazine*, 2(11):559–572, 1901.

T. J. Perkins and A. G. Barto. Lyapunov design for safe reinforcement learning. *Journal of Machine Learning Research*, 3:803–832, 2002.

L. Peshkin and E. D. de Jong. Context-based policy search: Transfer of experience across problems. In *ICML-2002 Workshop on Development of Representations*, 2002.

D. A. Pomerleau. Efficient training of artificial neural networks for autonomous navigation. *Neural Comput.*, 3:88–97, 1991.

S. Quinlan and O. Khatib. Elastic bands: Connecting path planning and control. In *IEEE Int. Conf. on Robotics and Automation (ICRA)*, pages 802–807, 1993.

N. Ratliff, B. Ziebart, K. Peterson, J. A. Bagnell, M. Hebert, A. K. Dey, and S. Srinivasa. Inverse optimal heuristic control for imitation learning. In *Proc. of AISTATS*, pages 424–431, 2009a.

N. Ratliff, M. Zucker, A. Bagnell, and S. Srinivasa. Chomp: Gradient optimization techniques for efficient motion planning. In *IEEE Int. Conf. on Robotics and Automation (ICRA)*, 2009b.

N. D. Ratliff, J. A. Bagnell, and M. A. Zinkevich. Maximum margin planning. In *26th Int. Conf. on Machine Learning (ICML)*, pages 729–736, 2006.

S. J. Raudys and A. K. Jain. Small sample size effects in statistical pattern recognition: Recommendations for practitioners. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 13:252–264, March 1991.

M. Riedmiller and H. Braun. A direct adaptive method for faster backpropagation learning: The rprop algorithm. In *IEEE Int. Conf. On Neural Networks*, pages 586–591, 1993.

S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 3rd edition, 2009.

T. D. Sanger. Human arm movements described by a low-dimensional superposition of principal components. *Journal of Neuroscience*, 20(3):1066–1072, 2000.

S. Schaal, J. Peters, J. Nakanishi, and A. J. Ijspeert. Learning movement primitives. In *International Symposium on Robotics Research*, pages 561–572, 2003.

M. Schmidt, G. Fung, and R. Rosales. Fast optimization methods for l1 regularization: A comparative study and two new approaches. In *18th European Conf. on Machine Learning (ECML)*, pages 286–297, 2007.

B. Schölkopf and A. J. Smola. *Learning with kernels : support vector machines, regularization, optimization, and beyond*. Adaptive computation and machine learning. MIT Press, 2002.

B. Schölkopf, A. J. Smola, and K.-R. Müller. Nonlinear component analysis as a kernel eigenvalue problem. *Neural Computation*, 10(5):1299–1319, 1998.

A. Shon, J. Storz, and R. Rao. Towards a real-time bayesian imitation system for a humanoid robot. In *IEEE Int. Conf. on Robotics and Automation (ICRA)*, pages 2847–2852, 2007.

B. Siciliano and O. Khatib, editors. *Springer Handbook of Robotics*. Springer, 2008.

J.-J. Slotine and W. Li. *Applied Nonlinear Control*. Prentice Hall, 1991.

E. Smith and S. Kosslyn. *Cognitive psychology: mind and brain*. Pearson/Prentice Hall, 2007.

A. J. Smola and B. Schölkopf. A tutorial on support vector regression. *Statistics and Computing*, 14:199–222, 2004.

S. Sonnenburg, G. Rätsch, C. Schäfer, and B. Schölkopf. Large scale multiple kernel learning. *Journal of Machine Learning Research*, 7:1531–1565, 2006.

F. Steinke, B. Schölkopf, and V. Blanz. Support vector machines for 3d shape processing. *Computer Graphics Forum*, 24(3):285–294, 2005.

M. Stolle and C. Atkeson. Transfer of policies based on trajectory libraries. In *IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS)*, pages 2981–2986, 2007.

J. Tegin, S. Ekvall, D. Kragic, J. Wikander, and B. Iliev. Demonstration-based learning and control for automatic grasping. *Intelligent Service Robotics*, 2:23–30, 2009.

R. Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society*, 58:267–288, 1996.

E. Todorov and W. Li. A generalized iterative LQG method for locally-optimal feedback control of constrained nonlinear stochastic systems. In *Proc. of the American Control Conference*, volume 1, pages 300–306, 2005.

M. Toussaint. Robot trajectory optimization using approximate inference. In *26th Int. Conf. on Machine Learning (ICML)*, pages 1049–1056, 2009.

M. Toussaint. Robotics. University Lecture, 2011. URL `http://userpage.fu-berlin.de/~mtoussai/teaching/11-Robotics/`.

M. Toussaint, N. Plath, T. Lang, and N. Jetchev. Integrated motor control, planning, grasping and high-level reasoning in a blocks world using probabilistic inference. In *IEEE Int. Conf. on Robotics and Automation (ICRA)*, pages 385–391, 2010.

I. Tsochantaridis, T. Joachims, T. Hofmann, and Y. Altun. Large margin methods for structured and interdependent output variables. *Journal of Machine Learning Research*, 6:1453–1484, 2005.

H.-H. Tu and H.-T. Lin. One-sided support vector regression for multiclass cost-sensitive classification. In *ICML*, pages 1095–1102, 2010.

B. Voelkl and L. Huber. Imitation as faithful copying of a novel technique in marmoset monkeys. *PLoS ONE*, 2(7):e611, 2007.

T. Wagner, U. Visser, and O. Herzog. Egocentric qualitative spatial knowledge representation for physical robots. *Robotics and Autonomous Systems*, 49(1-2):25 – 42, 2004.

V. Willert, M. Toussaint, J. Eggert, and E. Körner. Uncertainty optimization for robust dynamic optical flow estimation. In *The sixth Int. Conf. on Machine Learning and Applications*, pages 450–457, 2007.

F. Zacharias, C. Borst, and G. Hirzinger. Capturing robot workspace structure: representing robot capabilities. In *IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS)*, pages 3229–3236, 2007.

J. Zhang and A. Knoll. An enhanced optimization approach for generating smooth robot trajectories in the presence of obstacles. In *Proc. of the European Chinese Automation Conf.*, pages 263–268, 1995.

Zusammenfassung

Lernen von Repräsentationen aus Bewegungstrajektorien: Analyse und Anwendung in Roboterplanung und -Regelung

Ein Schwerpunkt der Forschung an Robotern ist die Generierung von Bewegung. Um die Gegenstände in seiner Umgebung handzuhaben und mit ihnen verschiedene Aufgaben zu lösen, muss ein Roboter die dafür notwendigen Bewegungtrajektorien berechnen und anschliessend ausführen. Eine Vielzahl Methoden zur Generierung von Bewegung wurde in der Robotik entwickelt. Doch keine dieser Methoden nutzt die Muster in vorangegangenen erfolgreichen Bewegungen aus, um die Bewegungsgenerierung besser und schneller zu machen. In dieser Doktorarbeit schlage ich neuartige Ansätze vor, die Techniken des Maschinellen Lernens und der Robotik vereinen. Diese Ansätze ermöglichen es einem Roboter, die Merkmale und latente Struktur in seiner Interaktion mit der Umgebung zu analysieren und dadurch aus seiner Erfahrung effizientere Bewegungen zu lernen. In zahlreichen Experimenten zeige ich, dass meine Ansätze schneller und effizienter Bewegungen erzeugen als etablierte Techniken.

Der erste konkrete Beitrag zu Bewegungsplanung ist die so genannte *Trajectory Prediction*. Unsere Methode kann, gegeben eine Weltbeschreibung, schnell eine geeignete Trajektorie, aufgrund einer von Daten gelernten Funktion, vorhersagen. Die von Daten gelernten Muster erlauben eine deutlich schnellere Bewegungserzeugung in Vergleich zu Methoden die diese Daten nicht benutzen. Diese Muster reflektieren Strukturen in der Anordnung von Objekte, die die robotische Bewegung beinflussen.

Mein zweiter Ansatz *Task Space Retrieval using Inverse Optimal Control* ist eine Methode zum Lernen aus der Beobachtung einer Bewegungsdemonstration. Aus der Demonstration lernt dieser Ansatz eine Wertefunktion für gute Bewegungen und eine kompakte Aufgabenrepräsentation. Dadurch kann ein Roboter auf Grundlage der gelernten Wertefunktion in neuen Situationen angemessene Bewegungen erzeugen und sich wie demonstriert verhalten, ohne dass ihm eine explizite Aufgabenspezifikation gegeben werden muss.

Vita
Nikolay Nikolaev Jetchev

CV not available in the online version due to privacy reasons.

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Dissertation selbstständig auf der Grundlage der in der Arbeit angegebenen Hilfsmittel und Hilfen verfasst habe.

Nikolay Nikolaev Jetchev
Berlin, den 31. Januar 2012