The Dissertation Committee for Reza Mahjourian
certifies that this is the approved version of the following dissertation:

Hierarchical Policy Design for Sample-Efficient Learning of Robot Table Tennis Through Self-Play

Committee:

Risto Miikkulainen, Supervisor

Sergey Levine

Aloysius Mok

Scott Niekum

Luis Sentis

Hierarchical Policy Design for Sample-Efficient Learning of Robot Table Tennis Through Self-Play

by

Reza Mahjourian

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

December 2018

Dedicated to my mother, father, and sister.

Acknowledgments

I would like to express my deepest gratitude to my supervisor, Risto Miikkulainen. Had it not been for his infinite patience and endless support I could not have completed this dissertation. His persistently positive, constructive, helpful, and forgiving attitude toward my work and work from other students in our group has been a tremendous resource and a source of inspiration. Risto's incredible writing skills have helped me find clarity in my work and have guided me to dig deeper into the problems at hand. Thank you Risto for not giving up on me! I also would like to thank my committee members Sergey Levine, Luis Sentis, Scott Niekum, and Al Mok, for their insightful comments, for guiding me toward important problems, and for their support at different stages of my work on this dissertation.

I would like to thank Martin Wicke, my first internship host at Google Brain, for his selfless support of my work and for sparking my interest in computer vision. Martin gave me a well-defined and interesting problem to work on, yet gave me space to develop my own ideas and approach. Despite his busy schedule, he always made time to meet with me and helped me with incredibly on-point research suggestions and technical solutions. His ability to empower and fully support his team to do their best remains a source of inspiration to me.

I would like to express my gratitude to the Google Brain Robotics team, especially Navdeep Jaitly, Vincent Vanhoucke, and Anelia Angelova for hosting me for more than a year during my work on this project. They provided me with a home at Google to do research and a sound board off of which I could bounce my ideas. I am eternally grateful for the opportunity to work with them during my stay. I am also grateful to have been able to work with Sergey Levine, who gave me great feedback and shared his deep insight on robotics and reinforcement learning with me. I am thankful to Erwin Coumans for creating the initial prototype for integrating the VR hardware. I would like to specially thank my co-author Nevena Lazic whose parallel experiments greatly accelerated my work. Thanks to David D'Ambrosio, Peng Xu, and Laura Graesser for their feedback on this work, and to other members of the Brain Robotics team for great discussions. Special thanks to Torsten Kröger for his help with using the Reflexxes library, and to Kurt

Hierarchical Policy Design for Sample-Efficient Learning of Robot Table Tennis Through Self-Play

Publication No. _____

Reza Mahjourian, Ph.D.
The University of Texas at Austin, 2018

Supervisor: Risto Miikkulainen

Training robots with physical bodies requires developing new methods and action representations that allow the learning agents to explore the space of policies efficiently. This work studies sample-efficient learning of complex policies in the context of robot table tennis. It incorporates learning into a hierarchical control framework using a model-free strategy layer (which requires complex reasoning about opponents that is difficult to do in a model-based way), model-based prediction of external objects (which are difficult to control directly with analytic control methods, but governed by learnable and relatively simple laws of physics), and analytic controllers for the robot itself. Human demonstrations are used to train dynamics models, which together with the analytic controller allow any robot that is physically capable to play table tennis without training episodes. Using only about 7000 demonstrated trajectories, a striking policy can hit ball targets with about 20 cm error. Self-play is used to train cooperative and adversarial strategies on top of model-based striking skills trained from human demonstrations. After only about 24 000 strikes in self-play the agent learns to best exploit the human dynamics models for longer cooperative games. Further experiments demonstrate that more flexible variants of the policy can discover new strikes not demonstrated by humans and achieve higher performance at the expense of lower sample-efficiency. Experiments are carried out in a virtual reality environment using sensory observations that are obtainable in the real world. The high sample-efficiency demonstrated in the evaluations show that the proposed

method is suitable for learning directly on physical robots without transfer of models or policies from simulation.[1]

Table of Contents

List of Tables

List of Figures

xvi

Chapter 1

Introduction

From ancient mythologies depicting *artificial people* to the modern science fiction writings of Karel Čapek and Isaac Asimov, there seems to be a clear image of what robots ought to be able to do. They are expected to operate in the world like human beings, to understand the world as humans do, and to be able to act in it with comparable dexterity and agility.[1]

Just as today most households can have personal computers in the form of desktops, tablets, and phones, one can imagine a future where households can use the assistance of humanoid robots. Rather than being pre-programmed to do specific jobs like communicating with people, helping with kitchen work, or taking care of pets, these robots would be able to learn new skills by observing and interacting with humans. They can collectively share what they learn in different environments and use each other's knowledge to best approach a new task. They already know their bodies well and are aware of their physical abilities. They are also aware of how the world and the common objects in it work. They just need to learn how to adapt to a new environment and a new task. If they need to learn a new skill by trying it out, they can do so efficiently. They can learn a lot from a few attempts and use reasoning and generalization to infer the best approach to complete the task without having to try it for thousands of times.

This dissertation takes a step in that direction by building a robotic table-tennis agent that learns the dynamics of the game by observing human players, and learns to improve over the strategy demonstrated by humans using very few training episodes where the agent plays against itself in a self-play setup.

---

[1]The body of all chapters in this dissertation is published in an article under the same title: Reza Mahjourian, Risto Miikkulainen, Nevena Lazic, Sergey Levine, and Navdeep Jaitly. Hierarchical policy design for sample-efficient learning of robot table tennis through self-play. *arXiv preprint arXiv:1811.12927*, 2018. All content except for experiments in Sec. 8.3 and Sec. 9.6 is contributed by the first author.

1.1 Motivation

The rate of progress in creation of intelligent robots seems to have been slower than other areas of artificial intelligence, like machine learning. That is because intelligent robotics requires not only human-like cognition, but also human-like movement and manipulation in the world. As of now, the most successful applications of robotics remain in the industrial domains, where the focus is on precision and repeatability. In those environment, the expected robot motion is known beforehand and there is no need to deviate from it. However, the general usability of robots depends on their ability to execute complex actions that require making multiple decisions through time.

Deep learning and reinforcement learning have been successful in solving interesting problems like object detection, playing Atari games, and playing board games like chess and Go. These advances have made it possible to approach human-level perception and cognition abilities. While perception problems can be learned in data centers using millions of data samples and training episodes, learning general robotic skills requires interacting with physical robot bodies and environments, which cannot be parallelized. Therefore, learning robotic agents need to be very efficient in how they use training samples.

This dissertation explores sample-efficient learning of complex robotic skills in the context of table tennis. Playing robot table-tennis games is a challenging task, as it requires understanding the physics of the robot and the game objects, planning to make contact with the ball, and reasoning about the opponent's behavior.

There have been many examples where application of deep learning to a problem has resulted in developing a superior approach with improved performance. For example, object classification and object detection tasks used to rely mainly on engineered SIFT features [11], an example of which is shown in Fig. 1.1. However, AlexNet [9] demonstrated end-to-end learning of object classification on ImageNet [7] using convolutional neural networks. Fig. 1.2 visualizes the convolutional filters learned in AlexNet in the first layer of the neural network. These filters can be regarded as the learned equivalents to SIFT image features. In the object classification domain, using neural networks to solve the task end-to-end allowed it to discover a suitable representation for image features that outperformed engineered features.

As another example, for the tasks of speech recognition and language translation, end-to-end learning has replaced the pipelines based on human-designed acoustic models, language models, and vocabularies with neural networks that outperform the old
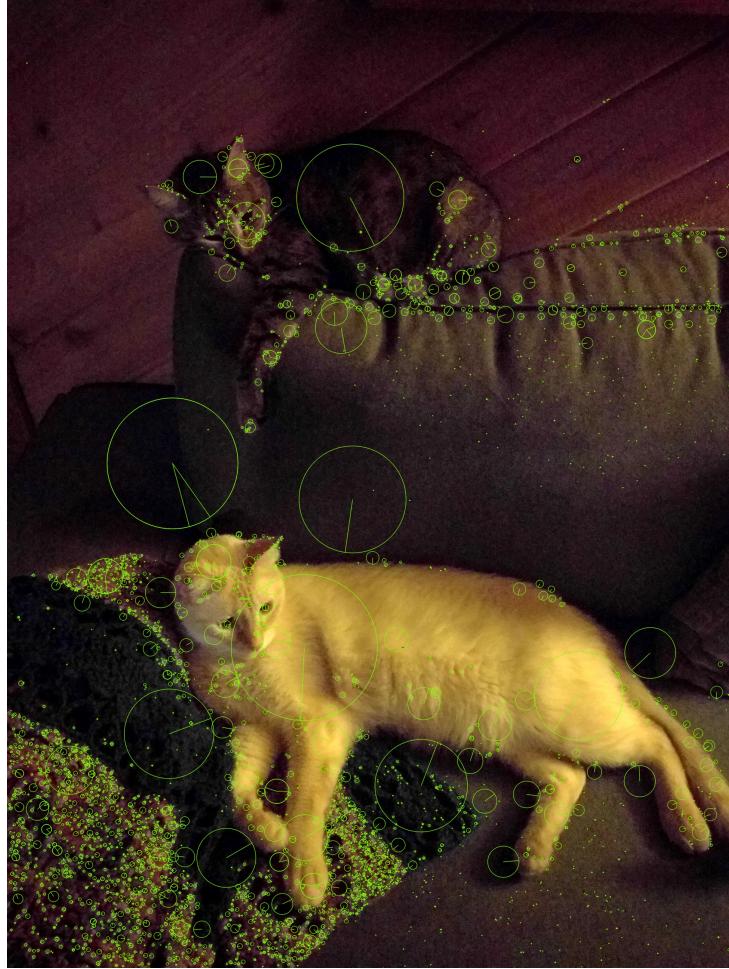
Figure 1.1: **Example SIFT Keypoints Detected in an Image.** SIFT keypoints can be used for object classification. The keypoints are extracted from images and individually compared to a database of existing keypoints extracted from other images. A matching algorithm can find candidate matching features based on the distance between feature vectors. Application of deep learning to object classification has resulted in discovering convolutional feature maps that are more effective than SIFT features.

approaches. In the classic pipelines, the vocabularies shared between the nodes were engineered and fixed. The components in the pipeline were restricted in choosing their outputs from the hand-designed vocabularies. In the end-to-end approach, the network is free to learn and use an internal embedding for the speech data and the language data.
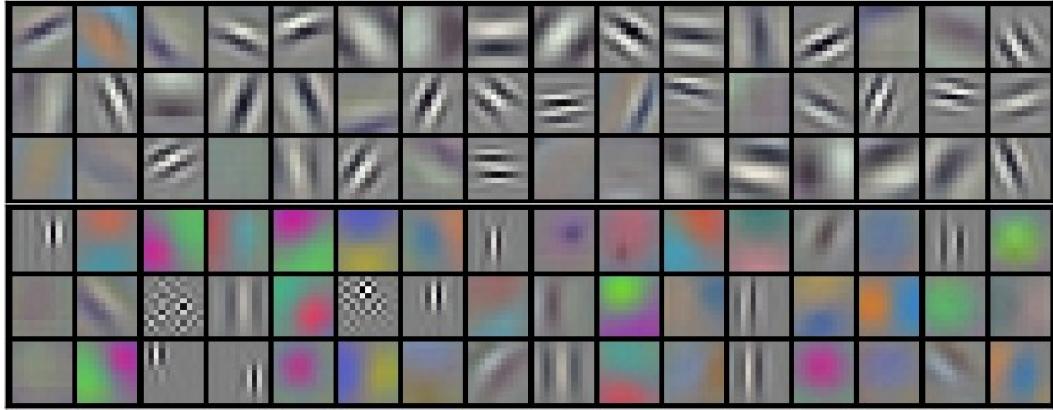
Figure 1.2: **Convolutional Filters Learned in AlexNet [9] for Image Classification.** The image shows 96 convolutional kernels of size 11x11x3 learned by the first convolutional layer in AlexNet. Deep learning is able to discover suitable features for the task of image classification. These learned features perform better than the engineered SIFT features. This example highlights the potential of learning algorithms to discover novel and effective solutions without a need for engineering.

This added freedom allowed deep learning to discover intermediate representations and features that are more suitable for solving the task.

Similarly, Mnih et al. [16] applied deep reinforcement learning to playing Atari games and demonstrated the ability of deep learning to discover a value network that can map raw pixels in the game to an expectation of future rewards.

These successes suggest that there is similar opportunity for applying deep learning to discover novel intelligent robotic behaviors. In the domain of table tennis, there is the potential for learning to discover:

1. **Better Strikes**: Can the robot swing the paddle in new ways beyond what humans have tried in table tennis? In sports, one can observe leaps where a player tries a new technique and then very quickly it is adopted by other players. For example, in the early nineties handball players started using spinshots that would hit the floor past the goalkeeper and turn to go inside the goal. Can reinforcement learning discover new striking motions for hitting the table-tennis ball?

2. **Better Game Strategy**: There are established human strategies for playing adversarial table-tennis games. Can reinforcement learning discover new overall gameplay strategies that are more effective in defeating a human opponent?

Discovering better motions and better strategies to solving tasks are relevant to household robots as well. This dissertation aims to utilize the ability of learning to discover such behaviors and demonstrate them in the domain of table tennis, and therefore show that such learning can be useful for general robotic tasks as well.

1.2   Challenges

General learning algorithms typically require millions of samples or training episodes to learn a task. Collecting samples for learning robotic tasks is costly, since each sample can cause wear and tear on the robot. The process is also time-consuming since interactions in the real world need to happen in real time and cannot be sped up by faster compute. In addition, robotic environments are often fragile and one cannot depend on agents learning automatically in unsupervised environments. Often, things break or objects get displaced requiring operator intervention to restore the setup to continue the training.

Moreover, there is usually an outer loop around the learning algorithms. Applying reinforcement learning is typically a trial-and-error process. The researchers usually develop new methods in an iterative manner by trying different approaches and hyperparameters. For every new instance of the problem, the learning algorithm is typically run from scratch. Therefore, in order to be feasible, advanced methods for learning general-purpose robotic tasks have to be able to use samples more efficiently than what is currently possible with deep learning and RL methods.

The end-to-end learning approach based on producing and consuming more and more data is not suitable for robotics. It is possible to bring some scale to learning robotic tasks using parallel hardware setups like arm farms. However, end-to-end learning methods often need so much data that this amount of parallelism is not enough to overcome the physical limitations that come with learning in the real world. An increase in the number of hardware setups also increases the expected frequency of hardware failures, which increases the need for human supervision.

Learning end-to-end policies poses another challenge, which is identifying the source of bugs or inefficiencies in one component of the implementation. In an end-to-end setup, the impact of a new change can only be observed by how it affects the overall performance of the system. Often learning algorithms are able to mask bugs by continuing to operate at a slightly reduced capacity or precision, thereby making it difficult to trace the root source of a problem after a few stages of development.

Some applications of deep learning to robotics can avoid some of the physical limitations by focusing on the perception part of the problem and ignoring learning motor skills. For example, object grasping can be approached as a regression problem, where the agent maps the input image to a grasp position and angle, which is then executed using a canned motion. However, when learning robotic skills it is very desirable for the learning algorithms to also discover novel motor skills. Learning algorithms may be able to discover new ways of handling objects that are more suitable for robots, and more effective with fewer degrees of freedom typically present in robot bodies.

A common approach to learning robotic tasks is sim2real: learning in simulation and then transferring the policy to work in the real world. With this method, learning can be done in the simulator. However, this approach requires solving a secondary problem, which is making the learned policy work with the real sensory observations and the control dynamics of the physical robot. Depending on the task, this transfer might not be any easier than learning the main problem.

Achieving breakthroughs in robotic learning most likely depends on discovering new learning approaches and new intermediate state and action representations that allow learning algorithms to spend a limited experimentation budget more strategically. Such intermediate state and action representations should be general enough to sufficiently capture the state of all policies. Yet, at the same time, they should be high-level enough to allow the learning agent to efficiently explore the space of all policies without having to try every combination. This dissertation takes a step in that direction by presenting an approach that achieves high sample-efficiency in learning the complex game of robotic table-tennis.

1.3   Approach

The approach presented in this dissertation offers a solution to the challenges discussed in the previous section by developing a learning solution that can discover

general robotic behaviors for table tennis, yet is **sample-efficient** enough that it can be deployed in the **real world** without relying on transfer learning from simulators.

The approach incorporates learning into a hierarchical control framework for a robot playing table tennis by using a model-free strategy layer (which requires complex reasoning about opponents that is difficult to do in a model-based way), model-based prediction of external objects (which are difficult to control directly with analytic control methods, but governed by learnable and relatively simple laws of physics), and analytic controllers for the robot itself.

The approach can be summarized around eight design decisions reviewed below. Using a *virtual reality environment* and collecting *human demonstrations* in this environment make the approach more **realistic**. Also, working with *low-dimensional state* instead of raw vision increases the chances that the sample-efficiency achieved in simulation would be reproducible in the real world. Introducing *rich high-level action representations* based on landing targets for the ball and target motion-states for the paddle, and *learning game-play strategies with self-play* enables the approach to discover **general** striking motions and versatile strategies for playing table-tennis. The division of tasks in a *hierarchical policy*, employing *model-based learning* for the striking skills, training the models over *low-dimensional state* and *high-level action representations*, and developing an *analytic robot controller* for executing high-level paddle-motion targets makes the method **sample-efficient**. The following subsections provide an overview of the main components of the approach.

1.3.1   Virtual Reality Learning Environment

In order to establish whether the approach can handle the complexity of real-world sensors, the method is developed in a Virtual Reality (VR) environment which allows for capturing the same sensory observations that would be available in a real-world table-tennis environment. Using VR instead of using plain simulation helps make sure the learning environment is realistic enough that the results would transfer to the real-world.

Although the method in this dissertation can be combined with sim2real approaches by using the models and policies learned in simulation as a starting point for training real-world models and policies, the emphasis in this dissertation is on developing an approach that would be sample-efficient enough to be able to learn the task from scratch in the real world. So, the method is developed in the VR environment, which has

more similarities to the real world. The next section describes how using low-dimensional state can increase the similarities to real-world setups. It also outlines how outlines how the chosen low-dimensional state can be obtained from physical sensors in the real world.

1.3.2   Using Low-Dimensional State

To make the environment more realistic, and to increase sample-efficiency, the approach uses low-dimensional state instead of raw vision. The observation and action spaces in the VR environment are chosen such that they have parallels in the real world. More specifically, the state space of the learning agents is limited to the low-dimensional ball-motion state and paddle-motion state. In the real world, ball-motion state can be obtained from ball-tracking algorithms, and paddle-motion state can be obtained with identical or similar sensors to what is used in the VR environment. Similarly, the action space of the striking policies is defined by paddle-motion targets, which can be tracked and controlled precisely on physical robots.

Ball-motion state includes the position and velocity of the ball. In the VR environment, ball-motion state is available from the underlying simulator. In the real world, a ball tracker [26, 4] can provide the position and velocity of the ball. Ball trackers usually track the ball velocity as well, since estimates on the current velocity of the ball can speed up the detection algorithm by limiting the search to a small region in the image and improve its accuracy by ruling out false positives. Detecting and tracking the location of a ball in a camera image is a relatively simple computer vision task. Ball tracking can be done with classic computer vision algorithms and does not require learning. The ball has a fixed geometry and a simple appearance in the camera images. A blob detection algorithm can identify the ball in the image. Given detections from two or more cameras and the camera intrinsics, the 3D location of the ball can be estimated. An advantage of using classic vision algorithms over using deep neural networks is the higher computational speed, which is critical in a high-speed game like table tennis.

Paddle-motion state includes the paddle's position, orientation, linear velocity, and angular velocity. When the paddle is attached to the robot, paddle-motion state can be obtained using forward kinematics. When learning from human games, paddle-motion state needs to be obtained from a motion-tracking system. There are a variety of solutions that allow for tracking the paddle-motion state with high accuracy. On the higher end, it is possible to use full-blown motion tracking systems to track marked and instrumented

paddles. On the lower end, one can use off-the-shelf tracking devices like HTC Vive, which can provide position information with sub-millimeter accuracy and jitter. Fig. 1.3 shows two types of VR trackers that work with HTC Vive. In fact, this is the same hardware that is used for experiments in this dissertation when collecting human demonstrations in the VR environment. Since such trackers are bulky, the human players would be able to use only one side of the instrumented paddles. Lastly, a more custom tracking setup can use small IMU sensors attached to the paddles. Visual markers on the paddles can be used to correct for the sensory drift that is common with IMUs.

(a) A Vive tracker.   (b) A Vive tracker attached to a paddle.
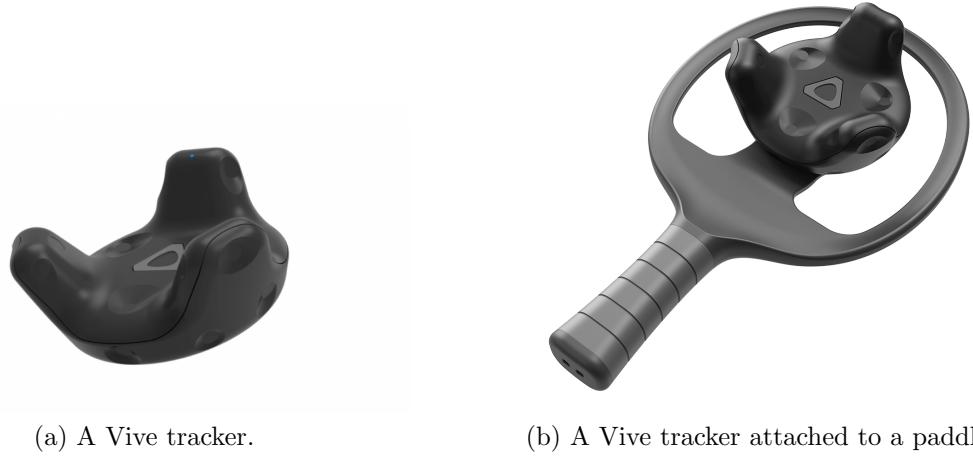
Figure 1.3: **Virtual Reality Trackers.** The trackers allow the position and orientation of objects to be tracked with sub-millimeter accuracy. In the VR environment, these trackers make it possible to capture the paddle motions generated by human players. The same trackers, or any other motion tracking technology, can be used to track the motion of table-tennis paddles in the real world. Photo credits: HTC.

Grounding the models and policies in low-dimensional state reduces the dimensionality of the learning problems and improves sample efficiency. Moreover, employing a separate component for extracting the low-dimensional ball-motion state from visual inputs makes it possible to debug and fine-tune that component before integrating it into the implementation for the learning agent. In contrast, using raw visual input would create a large disconnect between the distribution of sensory observations that are typically available in simulation, and raw visual data available from cameras in the real world, thereby limiting the extent to which the simulation experiments can predict real-world

9

performance. Another issue with using raw vision is that working with vision in the real world requires carefully designing the training environment to capture different lighting conditions, backgrounds, etc. Any mismatches between the training and test environments would greatly disrupt the performance of policies trained with raw vision. The next section describes how dynamics models trained on low-dimensional observations can inform the agents to make better decisions.

1.3.3   Model-Based Learning

Learning policies directly by interacting with the environment may require too many training samples. Model-free RL agents often need to implicitly learn to predict the outcome of their actions by predicting how their actions changes the environment. The approach in this dissertation uses model-based learning to increase sample-efficiency.

The game of table tennis, despite being a complex and fast game requiring great skill to play, has relatively simple physics compared to other tasks like robot locomotion or object grasping. In table tennis, most of the time, the ball is travelling in the air where it is only subject to gravity, drag, and Magnus forces due to its spin. The ball experiences short contacts with two types of objects: the table, and the player paddles. If the dynamics of the ball's motion and contact are understood, it is possible to both predict the ball's future states and to control for it by picking the right paddle motion to execute the desired contact.

The method uses observations in the environment to train dynamics models that predict the future state of the ball due to its free motion and due to contact with the player paddle. Such dynamics models can inform the learning agents about the consequences of the actions they are exploring. In contrast, in end-to-end model-free learning approaches, the agents are required to implicitly learn how the environment works in order to best exploit it and increase their reward. By capturing the simple physics of table tennis in dynamics models the method allows the learning agents to focus on learning high-level behaviors, thereby improving sample-efficiency. The next section describes how these dynamics models are trained.

1.3.4   Learning from Demonstrations

Training the dynamics models requires data. However, if there is no policy to drive the robot to play table-tennis, there is no way to collect the required data. On

the other hand, the observations that are needed to learn the ball motion and contact dynamics are readily available from human games. There is no need to use a robot to collect samples for training the dynamics models. Similarly, there is no need for kinesthetic teaching. Moreover, capturing human demonstrations is a lot easier than operating fragile robot setups to collect data. So, the approach trains dynamics models from human demonstrations.

The behavior of the ball and the contact forces between the ball and the table or the paddle are the same whether the paddle is carried by a robot or a human player. Contrast this with a task like locomotion. As the agent learns new gaits, it starts experiencing new joint states and new contacts with the ground, requiring any contact models to be adjusted. In table tennis, one can study the ball's free motion and contact behavior just by observing human games in instrumented environments. While collecting robot samples is costly and time-consuming, human samples can be obtained easily and abundantly.

Intermediate players may come short in their ability to move quickly, or to control the paddle correctly to execute their desired shot, which would pose a problem if policies are trained directly from human actions. Such policies would be able to play table-tennis only as well as the humans providing the demonstrations. So, this method only trains dynamics models from the human demonstrations and allows the policies to choose more versatile actions beyond what is demonstrated. The dynamics models are independent of the learning agent and stay valid as the learner's policy changes. They dynamics models can be used predict the future states of the ball when it is moving freely, and when it is hit by the paddle. The models can also help predict how to hit a ball so that it lands at a desired target on the opponent's side of the table. In other words, they can be used to choose actions. The next section describes how a rich high-level action representation gives the learning agents the ability to make very high-level yet general decisions for how to strike the ball with the paddle.

1.3.5   Rich General High-Level Action Representations

Playing table tennis requires moving the paddle around all the time. At any time during a rally, the agent is either trying to strike the ball towards the opponent, or trying to place itself optimally so that it can hit the next ball successfully. Learning to play table tennis by continuously making decisions about how to move on every single timestep makes for a very difficult problem. The reward for winning a rally may come

only at the end of a long sequence of actions. An RL agent would require too many samples to learn the basic techniques of striking the ball and moving around in the game just from the reward signal. Therefore, the approach defines two rich high-level actions that allow the agent to make game-play decisions at a high-level without losing generality in behavior: ball landing targets, and paddle-motion targets. The actions are illustrated in Fig. 1.4 and discussed below.

1. **Ball landing targets**: A landing target for the ball specifies a target motion-state for the ball at the moment that it lands. In general, the target motion-state can encode the ball's position, velocity and spin. In the current implementation, it encodes the position and speed (magnitude of velocity). Such a high-level action allow the agent to specify a striking action by its desired outcome. Learning to return the ball by specifying such targets is clearly easier than learning to return the ball by controlling the robot joints. At the same time, such actions do not reduce from the generality of policies. During each exchange between two players, each player's game-play can be represented by how the player lands the ball on the opponent's side of the table. No matter what movements the player executes, the effective action from the player is how they hit the ball and how that ball behaves after contact. In particular, the behavior of the ball at the moment of contact can fully capture the ball's behavior, as the ball's motion-state at that moment fully specifies its expected motion after contact with the table. So, although ball landing targets are high-level and easier to learn, they can capture and represent all possible striking behaviors (provided the implementation includes spin as well.)

2. **Paddle-motion targets**: A paddle-motion target specifies the target motion-state for the paddle at the moment that it hits the ball. Paddle-motion targets are an alternative action representation for parameterizing a strike. It is easier to learn to strike the ball by deciding a one-time target for the paddle than by deciding targets for robot joints over multiple timesteps. At the same time, paddle-motion targets are general action as well. During each strike, the impact of a player's paddle on the ball depends only on the motion-state of the paddle during the short period of time when the paddle and the ball are in contact. All the actions taken by the players up to the moment of contact are just in service to achieving a paddle-motion state at the moment of contact with the ball. So, representing a strike by the paddle-motion target that it achieves at the moment of contact does not reduce from generality
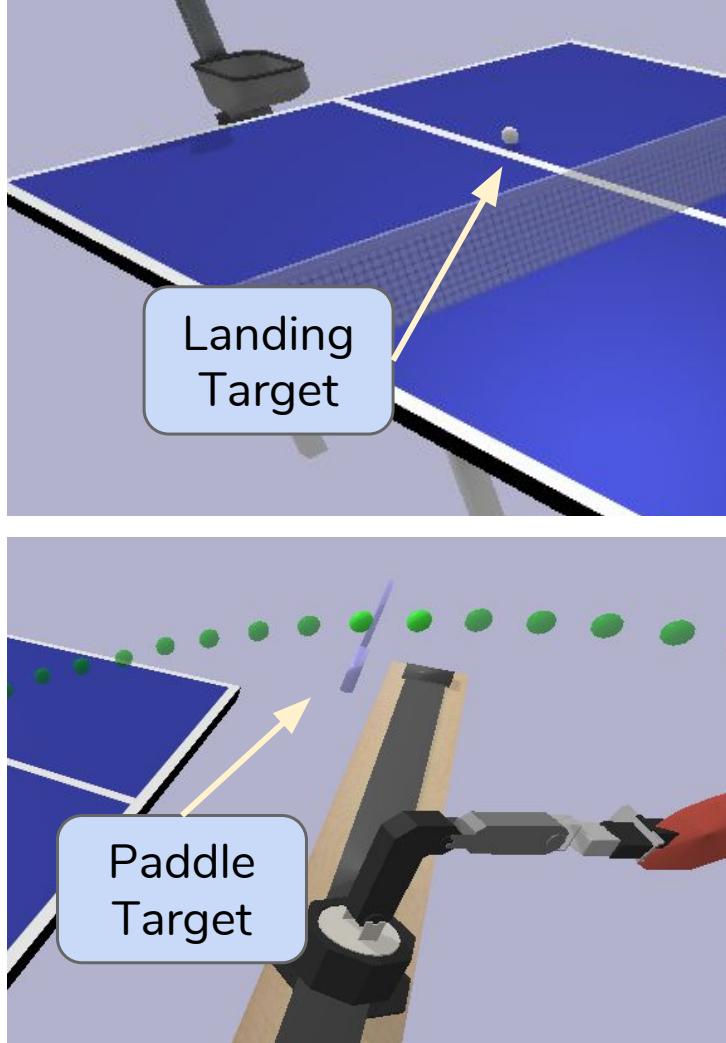
Figure 1.4: **Rich General High-Level Action Representations** Top: A ball landing target specifies the desired position and speed for the ball as it lands on the opponent's side of the table. Bottom: A paddle-motion target specifies the desired position, orientation, and velocity of the paddle at a desired time. In the illustrated example, the paddle target is chosen to intersect with the predicted trajectory of the ball over multiple timesteps, which is shown by the string of green balls. These high-level actions allow the agent to make high-level decisions about its game-play without losing generality in behavior. Therefore, they increase sample-efficiency.

of behavior. Paddle-motion targets can also be used to position the robot when it is waiting for the opponent to act. In those situations, the pose of the paddle is used as a proxy to control the position of the robot. Since the paddle is the main vehicle for the robot to play the game, this action representation is very suitable for deciding positioning targets for the robot during the waiting intervals in a rally.

Playing table tennis requires returning the ball to the opponent's side during each exchange. However, there are so many choices in returning the ball. The action representations used in this approach encode different striking behaviors, which permit different game-play strategies. Moreover, using such abstract action representations simplifies the action space for the agents. Since the actions capture the agent's behavior over multiple timesteps, they facilitates learning by eliminating the reward delay problem where a learning agent needs to figure out the actual action that leads to receiving a reward multiple timesteps into the future. Therefore the rich high-level actions increase sample-efficiency while maintaining generality in behavior. The following section describes how paddle-motion targets can actually be executed on the robot.

1.3.6 Analytic Paddle-Control

The dynamics models can inform the agent how to strike the ball, and the rich action representations allow the parameters of the strike to be specified at a high conceptual level. However, there needs to be a controller that can actually execute such high-level actions like paddle-motion targets. The method uses an analytic robot controller that is able to execute paddle-motion targets using information about the kinematics of the robot.

Executing paddle-motion targets requires precise control of the robot so that the paddle reaches the target position at the desired time, and that it has the desired velocity when it is crossing the target position. In addition, reaching the target requires executing a long sequence of joint-control commands that span over multiple timesteps. Learning to control the paddle by controlling the robot joints directly is a difficult task, as it requires learning implicitly how the robot commands affect the motion of the end-effector. So, using RL to learn to control the execute paddle-motion targets may require too many samples.

This method develops an analytic paddle controller which uses the Reflexxes trajectory-planning algorithm to execute any paddle-motion target from any starting

state for the robot, provided the target is achievable under the robot's physical limits and motion constraints. The Reflexxes library is able to compute optimal trajectories to reach the target motion-state while satisfying velocity, acceleration, and jerk limits for each robot joint. Employing this analytic controller removes the need for learning a paddle-control skill and improves the sample-efficiency of the method. The next section describes how the analytic controller together with other skill policies are used in a hierarchical policy to play the whole game of table tennis.

1.3.7   Hierarchical Policy

Playing table tennis requires technical skills in moving the paddle and striking the ball, and tactical skills in choosing appropriate targets at different points of a rally. This complexity is challenging for general-purpose RL algorithms. Therefore, instead of approaching table tennis as a monolithic task, this approach uses a hierarchical policy that decomposes table tennis into a hierarchy of subtasks. The hierarchical policy decouples the high-level skills from low-level skills in the game of table tennis, which makes it possible to implement each skill using a different method that is more suitable for it. Moreover, the hierarchy allows each skill to be developed, evaluated, and debugged in isolation. If necessary, the skills can be given perfect observations and perfect actions to fully evaluate their individual limits and errors. Such a setup allows for identifying and addressing inefficiencies in each component of the system before they are integrated and fine-tuned together as a whole.

In the task hierarchy, low-level skills like how to move the paddle to a target position are implemented using analytic controllers that do not require learning. Mid-level striking skills are implemented using dynamics models that are trained from human demonstrations with supervised learning. Lastly, the top-level strategy skill is trained with reinforcement learning, allowing the agent to discover novel behaviors.

In contrast, learning a task end-to-end may cause the model to relearn the primitive skills over and over in various states in presence of changing inputs. In other words, an end-to-end approach needs to learn to properly generalize its behavior to invariant states. Doing so requires more training episodes.

As explained in Sec. 1.3.5 the action spaces used in the task hierarchy are such that they do not reduce from the generality of the policies. In other words, the hierarchical policy does not restrict the agent's ability to explore the space of all possible game-

play strategies and techniques. As will be explained in Chapter 9, model-based policies employing human data can be more sample-efficient, while model-free policies that directly pick paddle-motion targets as actions can exhibit more novel striking motions at the expense of lower sample-efficiency.

The hierarchical policy permits learning general policies in a sample-efficient manner. The next section describes the hierarchical policy can discover interesting high-level game-play strategies.

1.3.8   Learning Strategy with Self-Play

The hierarchical policy design permits efficient training of general and parameterized low-level and mid-level skills which can execute different targets. However, driving these skills requires a game-play strategy. It is not possible to solve the game strategy analytically, as there are many choices in how to play table tennis and an effective strategy needs to factor in the behavior of the opponent as well. So, the approach trains the strategy skill at the top of the hierarchical policy using a model-free RL algorithm, that is free to explore the space of all possible game-play strategies with no requirements other than maximizing the reward.

Since at the beginning there are no opponents to play against, the approach uses self-play to train the agent against itself. As more self-play games are played, the strategy policy learns to adapt and respond to its own behavior. The strategy policy picks stochastic actions that set goals for the mid-level skills in the hierarchy. The strategy policy is encouraged to explore using an entropy term that rewards policies with more randomness.

The strategy skill allows the agent to make high-level decisions about its game plan without being concerned about how they are executed. The strategy skill is the only skill in the task hierarchy that requires exploration and uses reinforcement learning to train. By focusing the learning and exploration on this skill only, the method allows the agent to discover interesting general game-play strategies.

1.4   Guide to the Reader

The remainder of this dissertation is organized as follows. Chapter 2 describes the simulation and virtual reality environment. Chapter 3 provides a more in-depth overview

16

of the method than what is given in this section. Chapter 4 explains the hierarchical policy design and the subtasks in the task hierarchy. Chapter 5 describes how the learning environment is partitioned into a game space and a robot space so that the individual skills in the hierarchy can be trained with higher sample-efficiency. Chapter 6 explains the dynamics models that allows the agents to predict the future states of the game, and evaluates the predictive ability of the trained dynamics models. Chapter 7 describes the analytic paddle controller that is responsible for executing high-level paddle-motion action, and describes the implementation of the positioning policy. Chapter 8 describes the implementation of the different model-based striking policies and evaluates them against baseline model-free implementations that learn the striking skill from scratch using RL. Chapter 9 uses self-play to train table-tennis game-play strategies in cooperative and adversarial games. Chapter 10 provides a discussion on the work presented in this dissertation and outlines steps for future work, including how the method can handle vision and observation noise with continuous closed-loop control. Chapter 11 discusses related work on robotic table-tennis and hierarchical RL and provides a short review of the underlying learning method used in this work. Finally, Chapter 12 lists the contributions made in this work and concludes the dissertation.

Chapter 2

Simulation and Virtual Reality Environments

This chapter describes the simulation and Virtual Reality (VR) environment that is used for data collection, training, and evaluation of the table-tennis agent. First, the simulator and the virtual reality environment are introduced. Then, the reinforcement learning environment and its state and action spaces are described.

2.1   The Simulator

Fig. 2.1 illustrates the simulator's setup. The arm is mounted on a linear actuator, which allows the robot to move sideways. This configuration has a wider reach compared to a stationary robot. The linear actuator is implemented by one prismatic joint. The arm and the linear actuator are treated as one robot assembly with six joints. Fusing the linear actuator and the arm together in a single assembly simplifies inverse and forward kinematics calculations.

A different version of the simulation environment contains one robot playing against a table-tennis ball launcher. The ball launcher can shoot table-tennis balls with controlled initial conditions (position and velocity). By varying the initial conditions of every episode, the ball launcher makes it possible to explore the space of game conditions for the learning agents. This version of the environment is also used in evaluations.

The simulation environment is implemented on top of the PyBullet [5] physics engine. Simulation objects are defined by their geometries and their physics parameters including mass, coefficient of restitution (bounciness), friction coefficients, etc. The physics simulation in PyBullet is deterministic. So, there is no inherent noise in the simulation.

The physics are simulated at $1\,\mathrm{kHz}$. At each physics timestep, the object states and forces are recomputed and any collisions are recorded. Simulating physics at a high
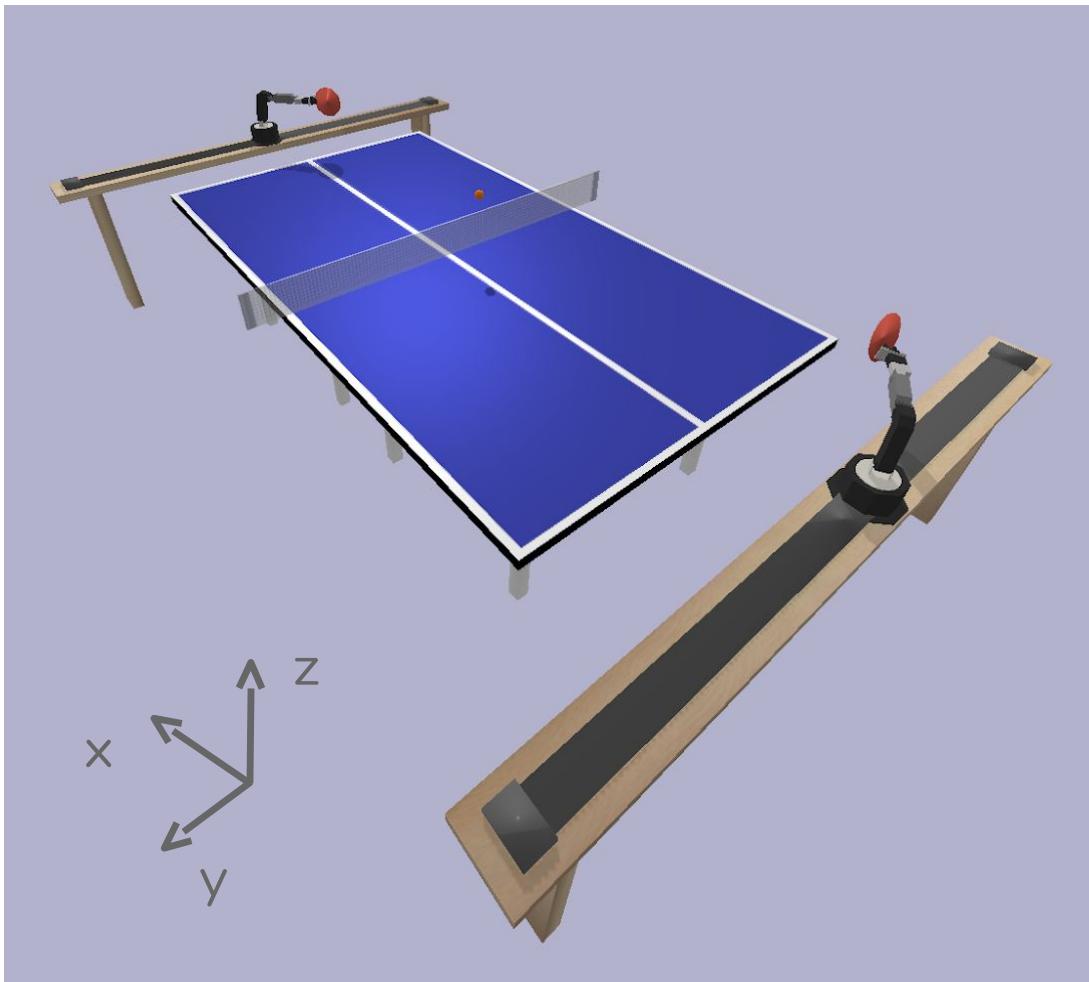
Figure 2.1: **Simulation Environment.** Two WidowX arms are mounted on linear actuators that allow the arms to move sideways. The two robot assemblies are at the same height as the table. The robot assembly consists of a linear actuator and a robot arm. The arm shown in the image is a WidowX arm with five joints. The original arm has a gripper, which has been removed in this setup and replaced with a fixed link holding a paddle.

frequency increases the fidelity of the simulation and avoids glitches like missed collisions due to the fast motion of the table-tennis ball.

2.2 Virtual Reality Setup

The simulator described in the Sec. 2.1 is connected to a virtual reality setup, allowing a human player to control a free-moving paddle. Using the VR setup makes it possible to create an immersive game environment where human demonstrations can be captured. The VR environment is a good proxy for capturing human demonstrations in the real world with instrumented paddles. In fact, the same trackers that are used in the VR setup can be used to instrument real table-tennis paddles and track their motion. This setup for capturing the human demonstration data makes it more likely that the methodology and the results would transfer to the real world.

The VR setup uses an HTC Vive headset, a controller (a.k.a. tracker), and two lighthouses. The components are shown in Fig. 2.2. The lighthouses continuously track the position and orientation of the player's headset and the controller in the player's hand. The HTC VR hardware uses active lighthouses and passive headset and controllers. The lighthouses emit vertical and horizontal sweeping laser lights at a fixed frequency. The headset and the controller have an array of light-detecting sensors that fire whenever they receive the laser light. Since the configuration of the sensors on the headset and controller are known, the timing of light-detection events reported by the different sensors contains enough information to decide the 3D position and orientation of each tracker. As long as a tracker is exposed to one of the two lighthouses and a sufficient number of its light sensors are visible to it, the device can be tracked with the same accuracy. So, if the paddle or the player hide the tracker from one of the lighthouses, it does not pose a problem. Fig. 1.3 shows two types of VR trackers that work with HTC Vive.

2.3 Learning Environment

The learning environment is implemented using the OpenAI Gym [3] API. The environment encapsulates the simulator and exposes the simulation object states as the environment state. At every timestep t, the environment exposes the following information on the objects:

- The ball-motion state b_t, which includes its 3D position $l(b_t)$, and velocity vector $v(b_t)$;

- The paddle-motion state p_t, which includes its 3D position $l(p_t)$, orientation $r(p_t)$, linear velocity $v(p_t)$, and angular velocity $\omega(p_t)$;
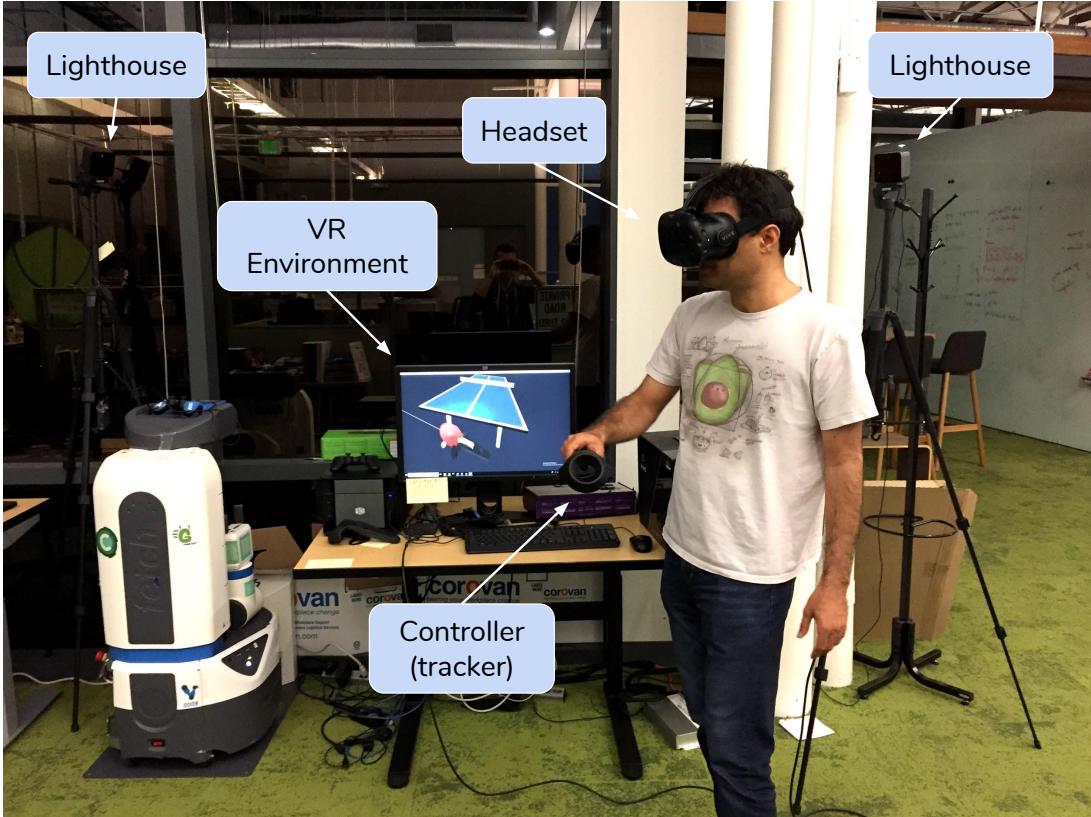
Figure 2.2: **Virtual Reality Setup.** A person is using the VR environment to play table tennis against a ball launcher in simulation. The VR hardware consists of two lighthouses, a headset, and a controller (tracker). The simulator is connected to the VR setup, such that moving the VR controller in the real world moves a floating paddle in the simulator. The paddle is used to hit the ball thrown by the launcher and return it to the other side of the table. The VR environment permits capturing realistic paddle motions as demonstrated by humans. The demonstrations are used to train dynamics models, which are then used by robotic agents playing table tennis against the ball launcher, or against each other.

- Robot joint positions q_t, and velocities $\dot{q}_t$;

- Most recent collision and the ball's location and velocity at the moment of collision.

Each learning agent defines its own observation space, which is a subset of the

environment state. The action space of the environment includes the six robot joints. The simulator supports position and velocity control modes for actuating the robot assembly.

There are three frequencies operating in the environment. The simulator runs at 1 kHz, allowing for smooth simulation of physics and control of the robot. The learning environment has a frequency of 50 Hz. Every 20 ms, the environment state is updated based on the most recent physics state of the objects. Collisions that are detected in between two environment timesteps are accumulated and reported together. The collisions contain no time information, so they appear to have happened at the end of the environment timestep. The high-level agents operate at a lower frequency in the environment. They receive observations and choose actions only once during each ball exchange between the players. Running at a lower frequency makes learning easier for the agents. The high control frequency is appropriate for smooth control of the robot, but the agents do not need to make decisions at every simulation or environment timestep. The lower frequency shortens the reward delay between the time the agent makes a decision and when it observes the consequence.

2.4   Conclusion

This chapter described the simulation and virtual reality environments that are used for simulating table-tennis games. The next chapter provides an overview of the proposed method for learning to play table tennis with high sample efficiency.

Chapter 3

Method Overview

This chapter gives an overview of the approach and its key components. It depicts a high-level picture of how the different components work together as part of the method. Sec. 3.1) discusses decomposing the task of playing table tennis into subtasks that can be learned or solved more efficiently. Sec. 3.2 describes decomposition of the environment to separate the problem of robot control from the table-tennis game-play. Sec. 3.3 discusses environment dynamics models that are trained from human demonstrations. Sec. 3.4 describes an analytic robot controller which can execute target paddle-motion states (pose and velocity). Sec. 3.5 discusses using self-play to learn high-level table-tennis strategies for cooperative and adversarial games.

3.1 Policy Design

Robot table tennis is a complex task, and therefore it may be difficult for reinforcement learning. The method decomposes the task into a hierarchy of skills where higher-level skills depend on lower-level skills. The low-level skills are easy to learn; in turn, exposing the functionality of these low-level skills as primitives to higher-level skills makes those skills less complex and easy to learn as well.

The task hierarchy, which is illustrated in Fig. 3.1, offers a high-level view of the task to the learning agents. Instead of continuously making decisions at every timestep, they make one high-level decision during each exchange with the opponent. The high-level decisions determine how to strike the ball when the agent is returning a shot, and how to position the robot when the agent is waiting for the opponent. Fig. 3.1 shows three variants of the policy based on different striking skills. The three striking skills are:

1. **Land-Ball**: Given the state of the incoming ball, hit it such that the ball lands at a desired location on the opponent's side with a desired speed.

2. **Hit-Ball**: Given the state of the incoming ball, hit it with a desired paddle orientation and velocity.

3. **Paddle-Control**: Given the state of the incoming ball, hit it at a desired position, with a desired paddle orientation and velocity.

The learning agents decide only targets for the striking skills. Using the striking skills as the action space for the agents eliminates the reward delay problem, and consequently, the agents require fewer training episodes to learn. At the same time, the striking skills does not reduce from the generality of policies. Any sequence of low-level actions can be reduced to, or represented by the state of the paddle and the ball at the moment of contact.
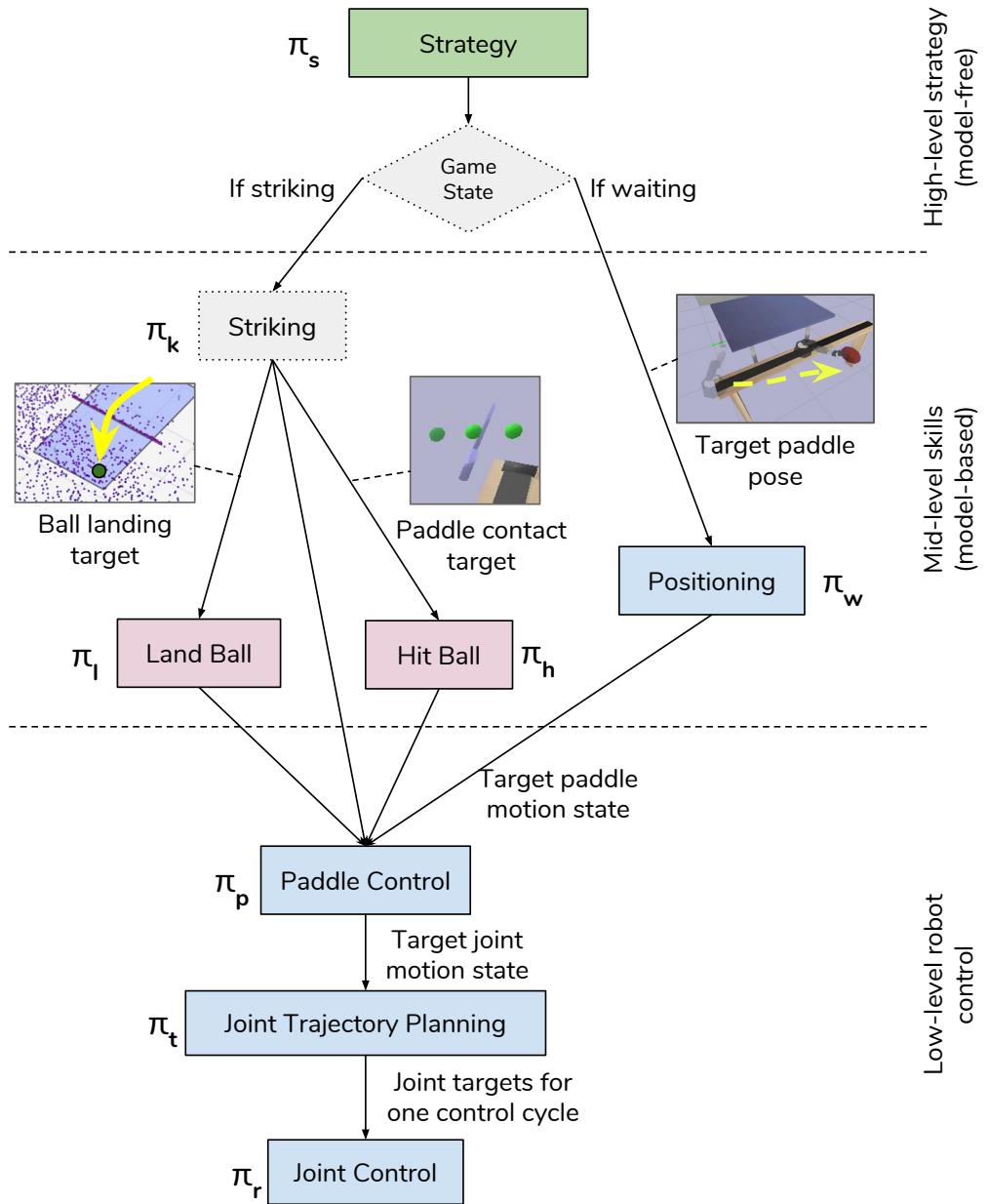
Figure 3.1: **The Skill Hierarchy.** The hierarchy consists of three levels of control (high-level, mid-level, and low-level), two modes (striking and waiting), and seven tasks. At any point in the game, the agent is either in striking mode or waiting mode. When in striking mode, the agent strikes the ball using one of three different skills: land-ball, hit-ball, or directly with paddle-control. Each variant of the policy uses only one of the three striking skills.

25

3.2 Environment Design

To make learning the skills in the hierarchical policy easier, the method decomposes the table-tennis environment into two spaces: the game space, and the robot space, as shown in Fig. 3.2. The game space is concerned with the physics of the table-tennis game involving a table, ball and paddles, independently of how the paddles are actuated. The robot space is concerned with the control of a robot that has a table-tennis paddle attached to the end of its arm. The game space includes only the table, the ball, and the paddle; it deals only with the game of ping pong. The robot spaces includes only the robot and the paddle; it deals only with the physics of the robot and end-effector control. The only object shared between the two spaces is the table-tennis paddle.

This separation makes it possible to study and model the physics of table tennis without any robot controllers or agent policies. In particular, it permits modelling the dynamics of the game just by observing humans playing table tennis with instrumented paddles. On the other hand, isolating the robot space makes it possible to focus on the problem of paddle-control without any complications from the game of table tennis.

Moreover, decomposing the environment makes it easier to replace the robot with a different model, since there is no need to retrain the game models from scratch.

3.3 Dynamics Models

Learning starts with observing human games with instrumented table-tennis paddles. The human demonstrations are used to train dynamic models for the environment. These models mainly capture the physics of ball motion and paddle-ball contact dynamics. Once such dynamics models are trained, they can be used to predict the future trajectory of the ball. They can also be used to make predictions about where a given ball will end up if it is hit with a given paddle pose and velocity.

In addition to predicting the outcome of actions, such dynamics models can be used for picking actions that can lead to desired states. For example, they can be used to decide how to hit an incoming ball to achieve a desired landing location and speed for the ball. In other words, they can be used to find the right paddle pose and velocity at the time of contact to achieve a desired land-ball target. The dynamics models reduce the land-ball task to a paddle-control task with a particular pose and velocity target for
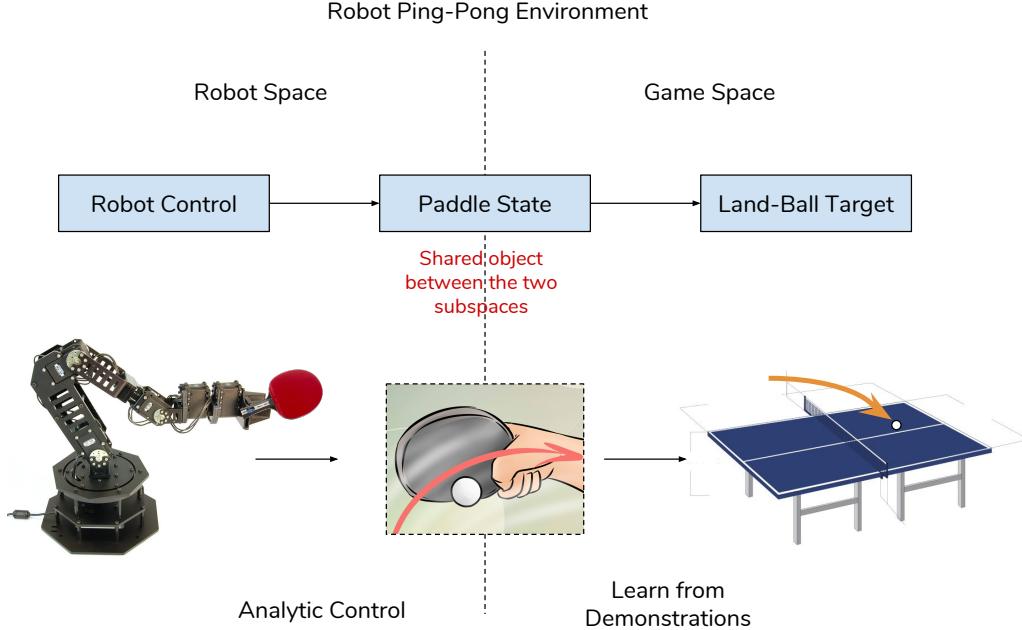
Figure 3.2: **Decomposition of the Robot Table-Tennis Environment.** The environment is divided into two spaces: the robot space and the game space. The robot space deals with the physics and control of the robot. The game space deals with the dynamics of table tennis. The only shared object between the two spaces is the table-tennis paddle. The decomposition makes it possible to learn the dynamics of table tennis from human demonstrations without using a robot. On the other hand it simplifies the robot control problem to just accurately controlling the paddle.

the paddle. Since the paddle needs to hit the ball, the paddle target also includes a time component.

3.4 Analytic Robot-Control

The task and environment decomposition simplify the control task in robot table tennis. Task decomposition reduces the game-play to accurate paddle-control and environment decomposition allows the robot controller to focus only on accurate paddle-control and not be concerned with the ball, table, or opponent.

Instead of relying on learning, the method relies mainly on analytic control to execute paddle targets. The target paddle pose is translated to target joint positions using inverse kinematics. The target paddle velocity is also translated to target joint velocities using the end-effector Jacobian. The Reflexxes Motion Library is used to computed an optimal trajectory starting with the current joint positions and velocities and reaching the desired joint positions and velocities at the desired time for contact. Using an analytic controller instead of learning increases the sample-efficiency of the method. It also makes it possible to switch the robot without having to retrain the paddle-control skill.

The method can thus produce table-tennis agents that follow fixed policies with minimal experimentation on the robot itself. Experiments are needed only to calibrate the motion constraints of the robot, and to model imperfections in the underlying robot control stack, e.g. imperfections in the robot's body and the PID controller.

3.5   Learning Strategy with Self-Play

The method uses a strategy skill whose job is to pick high-level targets for the striking and positioning skills. In a cooperative game where two agents try to keep a rally going for as long as possible, a good strategy might pick landing targets near the center of the opponent side of the table. In an adversarial game where the agents try to defeat the opponent, a good strategy might pick targets that make it difficult for the opponent to return the ball.

For tasks like land-ball and paddle-control, it is possible to evaluate any given action and determine whether it accomplishes the task. However, when looking at the whole game of table tennis and the space of all possible strategies, it is not immediately clear what action is more likely to help an agent win a point. It is exactly this part of the policy that benefits the most from reinforcement learning and evolutionary strategies and their ability to discover novel solutions.

This skill is trained with a self-play setup involving two robots. Training happens over a number of self-play levels. In the first self-play level, the agent plays against a fixed policy. In every subsequent level, the agent plays against a frozen copy of its most recent policy. Only one of the two robots is learning during self-play.

The strategy skill is the only skill in the hierarchy that requires every component to be in place. It requires all tasks controllers to be available. It also works across

the decomposition in the environment as it engages both the robot and game spaces. Despite these complex interactions, the strategy skill remains relatively simple, since its observation and action spaces are low-dimensional. Therefore, training the skill requires far fewer episodes compared to training an end-to-end agent.

A key challenge in learning with self-play is maintaining efficient exploration. Often with self-play learning the agent may converge to a narrow policy. Since the method uses self-play only for training the strategy, the exploration problem remains confined at the strategy level. A failure to fully explore the space of all strategies does not reduce the coverage of the dynamics models over the space of game physics, since they are learned independently. Also, a failure in exploration does not affect the analytic robot controller. In contrast, in an end-to-end setup a failure in exploration at the highest level may restrict the predictive ability of the underlying components as well.

The strategy skill also accounts for the imperfections in the underlying skills. The land-ball skill is not perfect. It misses the target by some error distance based on predictive accuracy of the dynamics models and the precision of the robot controller. During training, the strategy skill implicitly observes the accuracy of the underlying skills through its reward signal and learns to choose better targets for them. One common problem with task decomposition is that errors can accumulate through the task hierarchy. However, since the strategy skill does not have an externally-specified target in the method, it leaves room for the learning agent to compensate for the imperfections in the underlying skills.

3.6   Conclusion

To increase sample-efficiency in learning, the method decomposes both the task space and the environment space. Decomposition the task makes it possible to learn the skills one at a time. Decomposing the environment allows most skills to focus either on the dynamics of the game, or the dynamics of the robot. Therefore, task and environment decomposition increases sample-efficiency. At the same time, the decomposition is done in a manner not to reduce from the generality of the solutions. The following chapters describe each component of the method in detail.

Chapter 4

Policy Design

This chapter explains the different tasks that make up the control hierarchy in the method. The chapter starts with an overview of the task hierarchy. It then lists example inputs/outputs for some of the skills. Then, it discusses each task/skill in more detail. A high-level overview of the skill implementations are also provided. The terms task, subtask, and skill are used interchangeably. Often, skill is used when learning is required and task is used when an algorithmic controller is used.

The main advantage to task decomposition is to create subtasks that be implemented with different mechanisms. For subtasks that use some form of learning, decomposition improves sample-efficiency. Also, the training and debugging process is more manageable when focusing on one skill at a time. Decomposing the task makes it easier to develop and evaluate each component before it is integrated into the whole system.

4.1   Skill Hierarchy

The skill hierarchy is shown in Fig. 3.1. The hierarchy consists of three levels of control (high-level, mid-level, and low-level), two modes (striking and waiting), and seven tasks. At the top, there is the strategy skill, which just picks targets for the skills in the middle layer. Each skill in the hierarchy depends on skills below it and used the lower-level skills to accomplish its task. Except for the strategy skill, all other skills have parameterized targets. The striking and positioning skills provide an abstraction over the agent's plan for a single exchange with the opponent. The strategy skill, on the other hand, provides an abstraction over the agent's entire game plan by picking different striking and positioning targets over the course of multiple exchanges as the game goes on.

Decomposing the robot table-tennis task into smaller tasks makes learning easier and more sample-efficient. The green strategy node is learned with reinforcement learning.

The purple nodes have algorithmic policy implementations using dynamics models trained from human demonstrations. The blue nodes are implemented by an analytic robot controller.

At any point during the game, the agent is either in striking mode or waiting model. In striking mode, the agent uses a striking skill to hit the ball toward the opponent. The agent can strike the ball using land-ball, hit-ball, or directly using the paddle-control skill. Fig. 3.1 shows the three variants of the policy with alternative striking skills in the same image. In waiting mode, the agent picks a position for itself in anticipation of the opponent's next shot. This position is specified by a target paddle pose and executed using the positioning skill. The agent's mode changes automatically based on the current state of the game. As soon as the opponent hits the ball, the agent enters the striking mode. The agent stays in the striking mode until it hits the ball back. At that moment, it enters the waiting mode.

As discussed in subsequent chapters, the strategy skill is implemented by a PPO policy trained through self-play (Chapter 9). The striking skills land-ball and hit-ball are implemented by algorithmic policies (Chapter 8) which employ game dynamics models trained from human demonstrations (Chapter 6). The other four skills in the hierarchy are implemented using analytic control (Chapter 7). The remainder of this chapter defines each skill in detail.

4.2   Strategy

The strategy skill allows the agent to make high-level decisions about its game plan without being concerned about how they are executed. Depending on the state of the game, the strategy skill either picks a land-ball target or a positioning target and passes that target to one of its underlying skills. More specifically, the strategy skill is described by the policy

$$
\pi_s(b_s) = \begin{cases} \pi_k(b_s), & \text{if returning the ball} \\ \pi_w(l(p), \operatorname{sgn} N_x(p)), & \text{if waiting for opponent to return the ball,} \end{cases} \tag{4.1}
$$

where π_s denotes the strategy policy, b_s denotes the current ball-motion state[1], π_k denotes one of the three striking policies (defined later in this chapter), π_w denotes the positioning policy (defined later), $l(p)$ denotes a target position for the paddle, and $\text{sgn } N_x(p)$ denotes the sign of the x component of the paddle normal vector, specifying a forehand or backhand pose.

The strategy skill separates the tactical dimension of game-play from the technique needed to execute desired movements on the robot. By providing an abstraction over the game plan, it allows the agent to explore the space of game-play strategies while reusing the technical know-how captured in the underlying skills. The actions selected by the strategy skill stay active during one ball exchange, which lasts for a relatively long time interval (about 70-100 environment timesteps). So, it shortens the reward delay between the time the agent makes a decision and when it observes the consequence.

4.3 Striking Skills

During an exchange, when the agent is expected to return the ball coming from the opponent, it should execute some action to make contact with the ball and have it land on the opponent's side successfully. The method offers three variants of the striking policy which achieve this goal with different approaches. Each learning agent uses only one variant of the hierarchical policy, so for any given agent only one of the three striking policies is available. The three variants differ in the number of dynamics models they use in their implementation (ranging from three to zero) and are used in experiments to evaluate the impact of dynamics models on learning sample efficiency. The striking skill π_k is specified by one of the three policies

$$\pi_k(b_s) = \begin{cases} \pi_l(g \mid b_s) & \text{land-ball variant} \\ \pi_h(l_x(p_t), N(p_t), v(p_t), \omega(p_t) \mid b_s) & \text{hit-ball variant} \\ \pi_p(t, p_t \mid b_s) & \text{direct paddle-control variant,} \end{cases} \quad (4.2)$$

---

[1]In the current implementation, the strategy and striking skills receive only the current ball-motion state as the observation. Experiments have shown that adding other observations like the agent's paddle position $l(p_t)$ or the opponent's paddle position do not improve performance in this implementation. It is likely that in a setup where the agent can make more than one decision per exchange including such additional observations would be useful.

where π_l denotes the land-ball policy (defined later), π_h denotes the hit-ball policy (defined later), π_p denotes the paddle-control policy (defined later), b_s denotes the current ball-motion state, g denotes a ball landing target, $l_x(p_t)$ denotes the x coordinate of the paddle at time of contact (which maps to the desired distance from the net), $N(p_t), v(p_t), \omega(p_t)$ denote the normal, linear velocity and angular velocities for the paddle, t denotes the time of contact between the paddle and ball, and p_t denotes the full paddle-motion state.

Each of the three striking skills has a different approach to returning the ball. The land-ball skill parameterizes the strike by a desired landing target for the ball on the opponent's side of the table. Hit-ball parameterizes the strike by a desired paddle orientation and velocity at the time of contact with the ball. The hit-ball skill helps the agent make contact by automatically computing the required paddle position based on the predicted ball trajectory. Striking directly using the paddle-control skill requires fully specifying a target paddle-motion state at the time of contact. As described in Chapter 8, land-ball requires three dynamics models to work, while hit-ball requires only one dynamics model. The paddle-control skill does not use any trainable dynamics models. So the three alternative striking skills are used to evaluate the impact of using dynamics models on sample-efficiency of learning the different skills.

Sec. 4.3.1 and Sec. 4.3.2 describe the land-ball and hit-ball striking skills. Striking with direct paddle-control is covered in Sec. 4.5.

4.3.1   Land-Ball

The objective of the land-ball skill is to execute the necessary actions to hit back the opponent ball b_s in a way that it eventually lands at the desired target g. The target g specifies a target location $l(g)$, and a target landing speed $|v(g)|$ over the opponent's side of the table:

$$g = l(g), |v(g)| \tag{4.3}$$

The target landing location $l(g)$ specifies the desired position of the ball at the moment of landing:

$$l(g) = l(b_g) \tag{4.4}$$

33

where $l(b_g)$ denotes ball position at the moment of landing. Note that there is no constraint on the landing time. The subscript g here denotes some unspecified landing time. The target landing speed $|v(g)|$ specifies the desired magnitude of the ball's velocity vector at the moment of landing.

In the current implementation the landing target does not specify a desired landing angle for the ball. Since the trajectory of the landing ball is constrained at two points (by b_t and g), the landing speed specifies the landing angle to some extent, as faster balls tend to have lower landing angles. However, the implementation can be extended to also include a target landing angle. In environments where the ball spin affects it motion, topspin would increase the landing angle while backspin would decrease it. In such environments a desired landing angle can further constrain the desired ball motion.

An example land-ball target can be:

$$l(g) = (0.9, 0.2) \text{ m},$$
$$|v(g)| = 6 \text{ m/s}.$$

In a coordinate system where the center of the table's surface is at $(0, 0, 0.76)$ m, the two-dimensional target $(0.9, 0.2)$ specifies a point 0.9 m behind the net and 0.2 m to the left of the center divider (from the agent's perspective) on the opponent's side. The z coordinate of the target is always equal to 0.78 m, which is the height of a standard table, 0.76 m, plus the radius of a table-tennis ball, 0.02 m.

The land-ball skill is described by the policy

$$\pi_l(g \mid b_s) = \pi_p(t, p_t \mid p_s), \tag{4.5}$$

where π_p denotes the paddle-control policy (defined later), t denotes the planned paddle-ball contact time picked by the land-ball skill, and p_t denotes the target paddle-motion state at time t, also picked by the land-ball skill.

The implementation for the land-ball policy is described in detail in Chapter 8. Given the incoming ball's motion state p_s, the land-ball policy predicts the ball's future

trajectory and plans to make contact with it at some time t. The land-ball policy chooses a target motion state p_t for the paddle at the time of contact. To ensure contact with the ball, the target position for the paddle $l(p_t)$ is always chosen to be equal to the predicted position of the ball b_t at that time. The land-ball policy also chooses the paddle's target orientation, linear velocity, and angular velocity at the time of contact. To accomplish its requested goal, the land-ball policy should pick the paddle orientation and velocity such that hitting the ball with that paddle-motion state sends the ball to the landing target g. The target contact time t and the target paddle-motion state p_t computed by the land-ball skill are passed as a high-level action to the paddle-control skill.

The land-ball skill provides a high-level abstraction over the agent's game-play during a single exchange with the opponent. This high-level abstraction does not cause a reduction in generality of behavior. Barring deceptive movements to hide the agent's intention from the opponent, any sequence of paddle actions can be reduced to the resulting landing state for the ball.[2] In other words, the specification of the land-ball skill makes it possible to specify the agent's behavior by its desired outcome. Learning to use the land-ball skill is easier for the agents as its action space has fewer dimensions than a fully-specified paddle-motion state, yet its action space can specify complex behaviors. Furthermore, land-ball's action space maps to the geometry of the world and allows for exploiting the symmetries and invariances that exist in the domain (Sec. 6.3).

4.3.2 Hit-Ball

The hit-ball skill is an alternative striking skill. Rather than specifying the strike by how the ball should make contact with the table, the hit-ball skill specifies the strike by how the paddle should make contact with the ball. Unlike land-ball which aims for a particular point on the opponent's side, hit-ball has not specific target for when the ball lands. The hit-ball skill is described by the policy

$$\pi_h(l_x(p_t), N(p_t), v(p_t), \omega(p_t) \mid b_s) = \pi_p(t, p_t \mid p_s). \tag{4.6}$$

---

[2]A fully-specified landing state should capture the ball's 3D velocity and spin at the moment of landing. Since the simulator used in this dissertation does not model spin, the implementation uses a simplified representation for the landing state. However, the land-ball skill can be extended to accept more expressive landing targets.

The hit-ball skill helps the agent make contact with the ball by computing the time of contact t and the target paddle position $l(p_t)$ based on its inputs and the predicted ball trajectory. It expects that the agent provide the other contact parameters like the orientation of the paddle encoded by the paddle normal $N(p_t)$ and its linear and angular velocities $v(p_t), \omega(p_t)$. An example hit-ball target can be:

$$l_x(p_t) = -1.7 \text{ m},$$
$$N(p_t) = (0.97, -0.02, 0.22),$$
$$v(p_t) = (1.23, 1.19, 0.06) \text{ m/s},$$
$$\omega(p_t) = (-0.2, -0.05, 2.85) \text{ rad/s}.$$

In this example, the skill is requested to make contact with the ball when it is 1.7 m away from the net (10 cm in front of the robot).

The implementation for the hit-ball policy is described in detail in Chapter 8. It uses a model to predict the ball's future trajectory. It then considers the intersection of the predicted ball trajectory with the target contact plane $x = l_x(p_t)$ which is an input parameter to the hit-ball skill. The point in the predicted trajectory that is closest to this plane is picked as the target contact point. This point determines the contact time t and the full position of the paddle $l(p_t)$ at the time of contact. The other attributes of the paddle's motion state $N(p_t), v(p_t), \omega(p_t)$ are given as inputs to the hit-ball skill. Together with the computed paddle position, they fully specify the paddle's motion state p_t, which is passed as a high-level action to the paddle-control skill.

Unlike the land-ball skill which can only execute strikes demonstrated by humans, the hit-ball skill can be used to execute arbitrary paddle strikes. So a strategy trained over the hit-ball skill can better explore the space of paddle strikes. At the same time, as the experiments show learning with the hit-ball skill is less sample-efficient and requires more training time.

4.4 Positioning

The positioning skill is in effect in waiting mode – when the ball is moving toward the opponent and the agent is preparing to respond to the next ball from the opponent.

The objective of this skill is to move the robot to the requested positioning as as quickly as possible. Instead of specifying the requested position with the full robot pose, the skill accepts a target paddle position. This formulation makes the action space of the positioning skill independent of the robot and more likely to transfer to new configurations and new robots. The positioning skill is described by the policy

$$\pi_w(l(p), \operatorname{sgn} N_x(p)) = \pi_p(p \mid p_s), \tag{4.7}$$

where p denotes some paddle state that satisfies the requested paddle position $l(p)$ and normal direction indicated by $\operatorname{sgn} N_x(p)$. Unlike the land-ball skill which requests a specific time target for its desired paddle-motion state, the positioning skill does not specify a time. In this case, the paddle skill is expected to achieve the desired paddle state as fast as possible. In other words, it is expected to find some minimum time m and achieve the target paddle $p_m = p$ at that time. An example positioning target can be:

$$l(p) = (-2.13, 0.07, 1.02) \text{ m},$$
$$\operatorname{sgn} N_x(p) = +1 \text{ (forehand)}.$$

The positioning skill maps the requested paddle position and paddle normal direction to some robot pose that satisfies them. The positioning policy is discussed in detail in Sec. 4.4. In the current implementation, this skill requests a target velocity of zero for the joints at their target positions. However, a more complex implementation can arrange to have non-zero joint velocities at target to reduce the expected reaction time to the next ball coming from the opponent.

4.5 Paddle-Control

The objective of the paddle-control skill is to bring the paddle from its current state p_s to a desired target state p_t at time t. This skill is invoked both by the land-ball skill and the positioning skill. The paddle-control skill is described by the policy

$$\pi_p(t, p_t \mid p_s) = \pi_t(t, q_t, \dot{q}_t \mid q_s, \dot{q}_s), \tag{4.8}$$

where π_t denotes the trajectory planning skill (defined later), q_t denotes the target joint positions, $\dot{q}_t$ denotes the target joint velocities, q_s denotes the current joint positions, $\dot{q}_s$ denotes the current joint velocities. An example paddle target can be:

$$t = 0.72 \text{ s},$$
$$l(p_t) = (-1.79, 0.22, 1.05) \text{ m},$$
$$N(p_t) = (0.97, -0.02, 0.22),$$
$$v(p_t) = (1.23, 1.19, 0.06) \text{ m/s},$$
$$\omega(p_t) = (-0.2, -0.05, 2.85) \text{ rad/s},$$

where $l(p_t)$ denotes the target paddle position, $N(p_t)$ denotes the target paddle surface normal, $v(p_t)$ denotes the target paddle linear velocity, and $\omega(p_t)$ denotes the target paddle angular velocity. The paddle's angular velocity at the time of contact can be used to better control the spin on the ball. Although the PyBullet simulator does not support ball spin, controlling the angular velocity is useful in real environments.

The paddle-control skill chooses some paddle orientation to satisfy the requested surface normal. An example paddle orientation can be:

$$r(p_t) = 0.42\mathbf{i} + 0.45\mathbf{j} + 0.51\mathbf{k} + 0.58,$$

where $r(p_t)$ is a four-dimensional quaternion representing the paddle orientation and $\mathbf{i}, \mathbf{j}, \mathbf{k}$ are unit vectors representing the three Cartesian axes.

As discussed in Chapter 7, the policy for the paddle-control skill can be implemented analytically, thereby reducing the need for training and increasing the sample-efficiency of the method. The analytic controller uses inverse kinematics and the robot Jacobian to translates the target paddle-motion state into a set of joint positions and velocities $q_t, \dot{q}_t$ such that achieving those joint states brings the paddle to the state p_t. It

then passes the target joint positions and velocities to the trajectory planning skill. An example target joint state for a 6-DOF robot assembly can be:

$$t = 0.72 \text{ s},$$
$$q_t = 0.71 \text{ m}, (-1.31, 0.50, -0.35, -0.31, -0.17) \text{ rad},$$
$$\dot{q}_t = 1.06 \text{ m/s}, (2.23, -0.02, 0.19, 0.61, -0.09) \text{ rad/s}.$$

where the first DOF corresponds to the prismatic joint at the base of the robot assembly, and the next five DOFs correspond to the revolute joints in the arm.

The paddle-control skill provides a high-level abstraction over robot-control for table tennis. It allows higher-level policies to specify their desired robot actions by just specifying the impact of those actions on the paddle at a particular point in time. At the same time, this high-level abstraction does not cause a reduction in generality of behavior. Barring deceptive movements, any sequence of joint commands can be reduced to and represented by the final paddle-motion state at the time of contact. It is only during the short contact time that the state of the robot affects its impact on the ball. Lastly, since the paddle-control skill works with long-term actions lasting throughout one ball exchange, it allows the agent to observe the reward for its actions with little delay and learn more easily.

4.6 Joint-Trajectory Planning

The trajectory planning skill is responsible for bringing the joints from their current positions and velocities q_s and $\dot{q}_s$ to their target positions and velocities q_t and $\dot{q}_t$ at the requested time t, while observing the motion constraints on each joint. By doing so, it brings the paddle from its current state p_s to the target state p_t. The trajectory planning skill is used only by the paddle-control skill and is described by the policy

$$\pi_t(t, q_t, \dot{q}_t \mid q_s, \dot{q}_s) = \pi_r(\{(q_j, \dot{q}_j, \ddot{q}_j) \mid s \le j \le t\}),$$
$$\text{subject to:}$$
$$q_{\min} \le q_j \le q_{\max},$$
$$\dot{q}_{\min} \le \dot{q}_j \le \dot{q}_{\max}, \qquad\qquad (4.9)$$
$$\ddot{q}_{\min} \le \ddot{q}_j \le \ddot{q}_{\max},$$
$$\dddot{q}_{\min} \le \dddot{q}_j \le \dddot{q}_{\max},$$

where π_r denotes the joint-control skill (defined later), $q_j, \dot{q}_j, \ddot{q}_j, \dddot{q}_j$ denote the planned joint positions, velocities, accelerations, and jerks at times $s \le j \le t$, the motion constraints $q_{\min}, \dot{q}_{\min}, \ddot{q}_{\min}, \dddot{q}_{\min}$ specify the minimum joint positions, velocities, accelerations, and jerks, and $q_{\max}, \dot{q}_{\max}, \ddot{q}_{\max}, \dddot{q}_{\max}$ specify the maximum joint positions, velocities, accelerations, and jerks.

The trajectory planning task receives long-term targets in the future. In turn, it generates a series of step-by-step setpoints for the robot to follow on every timestep starting with the current time s and leading to the target time t. Each trajectory setpoint specifies target joint positions and velocities $q_j, \dot{q}_j$. In the implementation, setpoints are computed at $1\,\mathrm{kHz}$. So, for the example given in Sec. 4.5 the computed trajectory would contain 720 points over 0.72 s.

The motion constraints depend on the robot's physical properties, mass, maximum motor torques, etc, and would be different for each robot. The position limits depend on the configuration of the robot joints and links. The velocity and acceleration limits depend on the motors and their loads. Jerk limits are usually imposed to limit the wear and tear on the robot. The velocity and acceleration limits can be determined from the robot data sheets, or measured empirically. As explained in Chapter 7, trajectory planning is implemented using the Reflexxes Motion Library [10], which can evaluate the feasibility of trajectories and return the next step in under $1\,\mathrm{ms}$. Reflexxes assumes the motion constraints to be constant and independent of the current joint velocities. In reality, motors have limit profiles that vary based on their current load. Still, the limits can be averaged and tuned for the duration of typical motions in table tennis, which last about 0.5 s.

Once the trajectory is computed, it is sent to the robot controller to execute.

4.7 Joint-Control

The objective of the joint-control task is to execute the necessary control actions on the robot to follow the next setpoint in the trajectory. Typically this skill is already realized by an existing PID controller or an inverse dynamics controller. Then, the joint-control task simply represents the low-level controller on the robot. The task is described by the policy

$$\pi_r(\{(q_j, \dot{q}_j, \ddot{q}_j) \mid s \le j \le t\}) = \{u_j \mid s \le j \le t - 1\} \tag{4.10}$$

where u_j denotes the control action at time j.

At each point in time, the controller observes the current joint states $(q_j, \dot{q}_j, \ddot{q}_j)$, and decides the joint actions or torques u_j to best satisfy the requested next joint states at the next timestep $j + 1$.

4.8 Conclusion

The skill hierarchy breaks down the task of playing table tennis into a hierarchy of simpler subtasks that depend on each other. This breakdown makes it possible to focus on each skill individually and develop and evaluate them separately. It helps isolate the source of problems and reduces the effort needed for debugging. Also, learning subtasks can be done with fewer training samples than learning the whole task at once.

The skill definitions do not impose any restriction on how they should be implemented. In order to achieve higher sample-efficiency, the approach uses reinforcement learning and supervised learning strategically. The robot-control skills and the positioning skill are implemented using an analytic robot controller that does not require learning. The striking skills are implemented by algorithmic policies that use dynamics models trained from human demonstrations using supervised learning. Human games provide observations on the physics of how the ball moves, and how it is affected by contact with the table or player paddles. Using the demonstrations to train dynamics models instead of policies has the advantage that it allows the agent to exhibit different game-play strategies beyond what is demonstrated by humans. Since the strategy requires complex reasoning

about opponents and is difficult to learn in a model-based way, model-free reinforcement learning is used to learn the strategy skill.

The next chapter discusses how decomposing the environment simplifies the implementation of skills and improves the method's sample efficiency. The skill implementations are discussed in future chapters. Chapter 7 discusses an analytic controller that implements paddle-control, joint-trajectory planning, and joint-control skills, plus the positioning skill. The striking skills land-ball and hit-ball are discussed in Chapter 8. Finally, Chapter 9 discusses how model-free reinforcement learning is employed to discover creative table-tennis strategies in cooperative and adversarial games using the skills described in this chapter.

Chapter 5

Environment Design

The skill hierarchy offers a decomposition of the problem on the behavioral axis. The method also decomposes the robot table-tennis environment into two spaces: the game space, and the robot space. The decomposition of the environment is shown in Fig. 3.2. The game space is concerned with the physics of the table-tennis game involving a table, ball and paddles, independently of how the paddles are actuated. The robot space is concerned with the control of a robot that has a table-tennis paddle attached to the end of its arm. The robot space does not deal with the table-tennis ball or the table. The paddle is the only shared object between the robot space and the table-tennis space. Also, note that the game space and its constituent objects are exactly the same in robot games and human games, which makes it possible to learn the game dynamics from human demonstrations and use them for robot games.

5.1 Game Space

Fig. 5.1 illustrates the game space and visualizes some of the variables which are used in the definition of the skills in Chapter 4. The game space contains only the table, paddle, and ball. Separating the game objects from the robot makes it possible to learn the dynamics and physical behavior of the objects in the game of table tennis independently of any particular robot. The data that is relevant to the game space and the dynamics models that are trained in this space can be used by agents driving different robots. Separating the game physics from robot control increases the sample-efficiency of training skills and policies that depend on them.
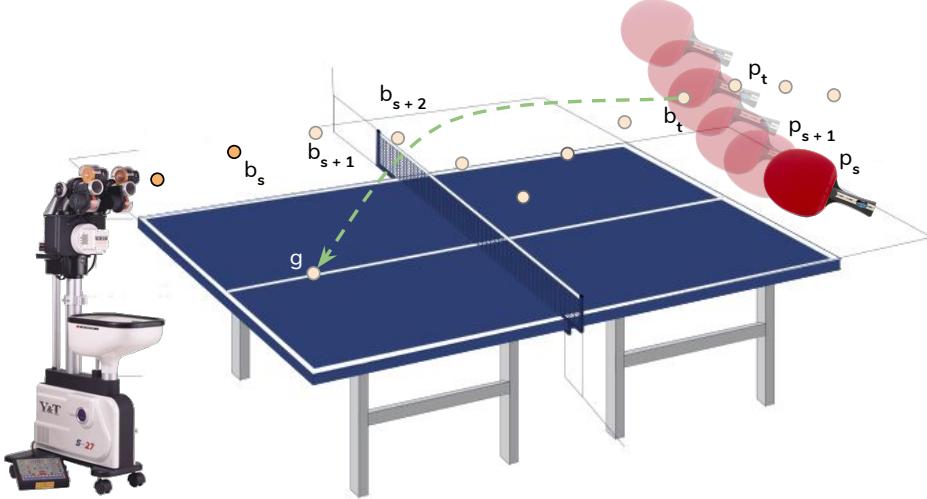
Figure 5.1: **The Game Space of the Environment.** The game space includes only the table, paddle, and ball. The variables b_s and b_t denote the ball motion-state at times s and t. Likewise, p_s and p_t denote the paddle motion-state at times s and t. The game space does not include any robot. Separating the game objects from the robot permits training dynamics models that predict the dynamics of the game itself independently of how the game is to be played by the robot. In particular, the game dynamics models can be trained from human demonstrations without using any robot.

5.2 Robot Space

The robot space deals with the control of the paddle attached to the robot's arm. Fig. 5.2 illustrates the robot space. The only entities that exist in this space are the links and the joints of the robot assembly and the paddle. The robot space is not concerned with any of the game objects, including the ball, the table, or any opponent. The main task in the robot space is paddle-control, which requires taking the paddle from its current motion-state p_s to a target motion-state p_t at time t. Separating the robot from the game objects simplifies the robot-control problem in the hierarchical policy, and reduces it to precise control of the end-effector to achieve target motion-states for the paddle.

44
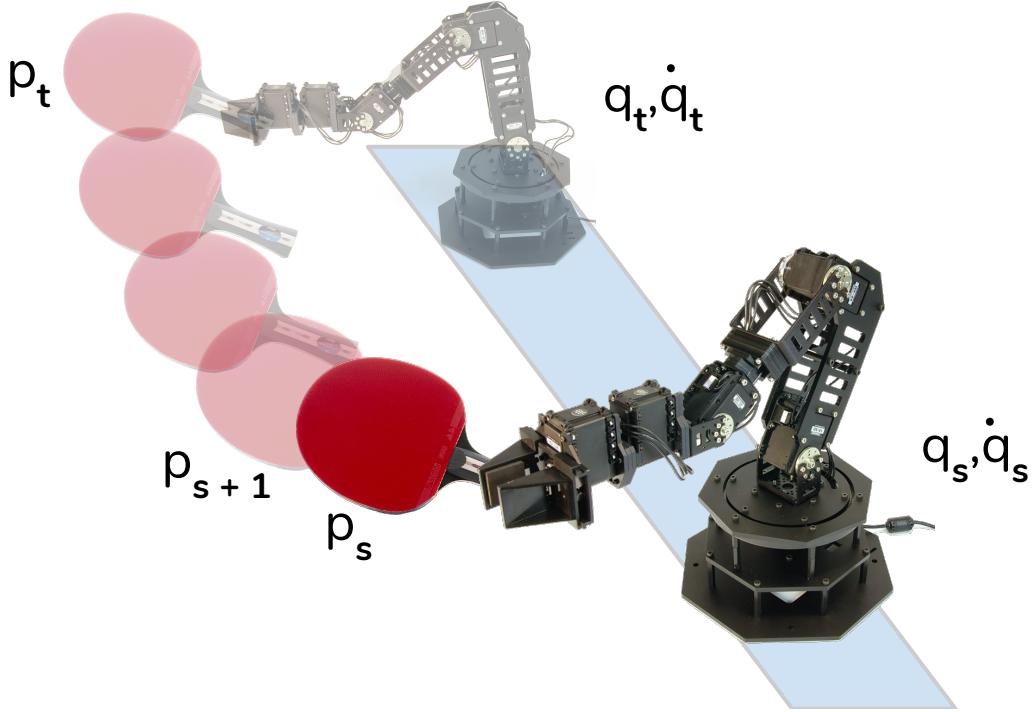
Figure 5.2: **The Robot Space of the Environment.** The robot spaces includes only the robot and the paddle attached to it. The main task in the robot space is paddle-control, which is defined by executing a paddle-motion target p_t starting from the current paddle-motion state p_s. The analytic controller maps the paddle-motion target to joint position and velocity target $q_t, \dot{q}_t$ and executes them using a trajectory-planning algorithm. The robot space does not include any of the game objects including the ball and table. Separating the robot from the game objects simplifies the implementation of the paddle-control skill. In particular, it permits solving the paddle-control problem with an analytic robot controller that does not need training.

5.3 Separating Physics of the Robot from the Physics of the Game

The key advantage of decomposing the environment is that it allows for separating the physics of the game objects from the physics of robot control. This decomposition makes it possible to create models of the interactions between the paddle, the ball, and the table independently of how the robot agent acts in and manipulates the environment.

Since the game space does not include the robot, it is possible to train dynamics models for it just by observing human games or human practice sessions against a ball launcher. Such models capture the physics of table tennis and can be used to predict the future states of the game given past observations. As shown later in Chapter 8, these models are used to create a functioning table-tennis agent without using any training episodes.

Furthermore, this decomposition simplifies the problem of robot control, as the robot space does not contain any elements of the game besides the paddle that is attached to the robot. When attached to the end of the robot arm, the paddle becomes the robot end-effector. Control of an arm end-effector is a well-studied problem in robotics, so the method uses classic control methods to develop an analytic controller for moving the paddle. In contrast, when the robot is part of the table-tennis environment, robot control becomes embedded in the agent's behavior. In such a monolithic environment, the table-tennis agent would need solve the robot control problems as well. For example, if the agent controls the robot in joint space, it would need to implicitly solve the inverse kinematics problem to be able to interact with objects in the environment. However, inverse kinematics has an analytic solution and does not require training samples. Using an analytic controller where possible decreases the complexity of the agent and increases sample-efficiency as no training samples are used by the agent to learn the internal dynamics of the robot.

5.4   One Game, Different Robots

An advantage of decomposing the environment into the game and robot spaces is that the robot can be replaced with a completely different type of robot without affecting the dynamics models that are trained on the game space. All that is needed to get the new robot to play table tennis is to supply a new robot controller that knows how to drive the new robot's end-effector properly. As long as the new controller can move the paddle attached to the robot as demanded, the new agent can continue to use the existing game dynamics models from the previous setup. The game strategy learned on an old robot might transfer to some extent to a new robot as well. However, since different robots have different reaches and different motion constraints, the strategy skill would need to be retrained for best exploit the physics of the new robot.

5.5   Reduction in Dimensionality

Another benefit to decomposing the environment is that it lowers the dimensionality of the inputs to the models and policies. In a monolithic environment, a typical choice for the observation space is to include the joint states in addition to the state of the world objects. Consider the problem of predicting the trajectory of the incoming ball. The future states of the ball only depend on the state of the ball and its impact with the table. Yet, in a monolithic environment, the joint states are always part of the observations. Similarly, the state of the robot paddle depends only on the joint actions, and is independent of the ball's position. On the other hand, when learning to land the ball on the opponent's table, the agent needs to line up the paddle with the ball. For this task, the agent needs to take into account both the state of the paddle and the state of the ball.

In a monolithic environment, more samples or more training episodes are required for the agent to learn when the dimensions are independent and when they interact with each other.

5.6   Interaction with Task Decomposition

In the method, environment decomposition is used in combination with the task decomposition in the skill hierarchy. In the skill hierarchy in Fig. 3.1, the strategy and land-ball skills interact mainly with the game space of the environment. The positioning skill, the paddle-control skill and the skills below it interact only with the robot space of the environment. However, the usefulness of decomposing the environment does not depend on using task decomposition as well. Even with an end-to-end policy which treats robot table tennis as one monolithic task, decomposition of the environment into game and robot spaces can facilitate learning.

In a model-based setup, decomposing the environment allows the agent to learn something about the robot and something about the game from each training episode. Regardless of how good the current policy is, each training episode provides the agent with a sample from which it can learn how the robot's body responds to the chosen actions. At the same time, the episode provides the agent with observations showing the dynamics of interactions between objects in the table-tennis environment.

As an example, consider the task of producing striking motions with the goal of

making contact with the ball. In a monolithic environment, as the policy improves and the agent learns to hit the ball, the distribution of the inputs to the dynamics models may change considerably. In a decomposed environment, the models that predict the impact of joint commands on the motion of the robot paddle will observe the same outcome regardless of whether the strike is successful at making contact with the ball or not. Suppose at some point the agent learns to make contact. The experience from before the agent's policy works well transfers fully to after when it learns to make contact.

5.7   Conclusion

Separating the environment and robot models creates a setup where the table-tennis dynamics models can be trained independently from the robot's control mechanism. This separation helps increase sample efficiency during training, as any shortcomings in the robot control stack would not limit the space of observations for the game dynamics models. Similarly, in the other direction, any shortcomings in the agent's table-tennis strategy would not hinder an exploration in the space of the robot's end-effector behavior. The next chapter describes the dynamics models that make predictions over the game space of the environment.

Chapter 6

Dynamics Models

This chapter describes the dynamics models that can make predictions over future states in the game space in the table-tennis environment. Sec. 6.1 describes some of the design choices behind implementing dynamics models as neural networks. Sec. 6.2 describes the ball-trajectory and landing prediction models. Sec. 6.3 describes the normalization process for reducing the number of dimensions for the dynamics models and thereby reducing the number of samples needed for training them. Sec. 6.4 describes the process for collecting training samples in the VR environment. Sec. 6.5 evaluates the trained dynamics models.

6.1 Learning Dynamics with Neural Networks

This section discusses why the human demonstrations are used in this work to train dynamics models rather than policies, and why the dynamics models are implemented as neural networks rather than physics models or a combination of the two.

6.1.1 Learning Dynamics Instead of Policy

Human demonstrations can be used to teach a learning agent about optimal actions to choose. In other words, the demonstrations can be used to directly train a policy. In this dissertation, the human demonstrations are used only to train dynamics models of the game space of the environment. It has the advantage that the dynamics models transfer to different policies. Also, not using the human policies allows the strategy agent to come up with novel approaches to playing table tennis different from what was tried by the humans.

6.1.2   Using Physics vs. Neural Networks

A good approach to implementing the dynamics models is to use a combination of physics models and neural networks. The physics models would include parameters for gravity, coefficients of friction, restitution, the drag and Magnus forces acting on the ball, etc. The values for these parameters can be obtained by computing a best fit over observations from the training data. Such physics models can produce an approximation over the future states of the objects. Then, neural networks can be trained to predict only a *residual* over the predictions from the physics models. The targets for the neural networks can be obtained by running more accurate offline state estimation models that take complete trajectories into account, including data points that are to be observed in future timesteps. Using neural networks in combination with physics models is expected to increase the sample efficiency.

However, since the method is evaluated only in simulation, all dynamics models are implemented using neural networks. The reason is that the physics simulations in Bullet are deterministic. If the right parameters were included in the physics models, their values could be discovered rather easily, without relying on the predictive ability of the neural networks. By relying only on neural networks, the experiments evaluate whether the dynamics models are able to model physical interactions. If they do so in the simulator, they are likely to do so in the real world as well.

6.2   Dynamics Models

The models are used to make two types of predictions:

1. Future trajectory of the current ball: Once the opponent hits the ball, the learning agent predicts the future trajectory of the incoming ball and picks a desired point and time of contact for hitting the ball back.

2. Landing location and speed resulting from a paddle action: The learning agent uses landing predictions to choose among potential paddle actions to best satisfy the requested landing target. The landing model predicts the eventual landing location/speed of the ball on the opponent side given the pre-contact ball-motion and paddle-motion states.

The following sections go over each dynamics model in detail.

6.2.1 Ball-Trajectory Prediction Model

The ball trajectory prediction model receives the estimate on the current state of the ball and predicts a sequence of ball positions and velocities at future timesteps. The model produces a full trajectory to give the agent multiple options for picking a contact point for hitting the ball with the paddle. The model is described by the function

$$b_{s+1}, b_{s+2}, \ldots, b_{s+n} = B(b_s), \tag{6.1}$$

where B denotes the ball trajectory prediction model, b_s denotes the estimate on the current ball-motion state, and $b_{s+1}..b_{s+n}$ denote predictions on the future ball-motion state over the next n timesteps.

The model's network architecture is shown in Fig. 6.1. It is a recurrent model, however, it receives the same input at every timestep. The model outputs form a complete trajectory for the ball's future position and velocity over multiple timesteps.

Given the training data, the model learns to predict the physical forces that operate on the ball. In the simulator, the free motion of the ball in the air is affected by its velocity, gravity, and a drag force that is proportional to the velocity of the ball. The Bullet simulator does not simulate ball spin and the Magnus forces causing the ball's trajectory to curve. The motion of the ball after contact with the table is affected by the friction coefficients (lateral, rolling, spinning) between the ball and table, and the restitution coefficients of the ball and table. In order to predict the future states of the ball, the neural network learn to implicitly model these physical forces. The ball trajectory may include arcs after its contact with the table. The training data allows the model to learn about the location and geometry of the table and predict whether the ball will collide with the table and how its trajectory is affected after contact.

6.2.2 Landing-Prediction Model

The landing-prediction model allows the agent to make predictions about the eventual landing location and speed of the ball given pre-contact ball and paddle-motion states. Suppose the land-ball agent has predicted a trajectory for the incoming ball and has picked a particular point b_t as a desired contact point for hitting the ball with the paddle. Given

51
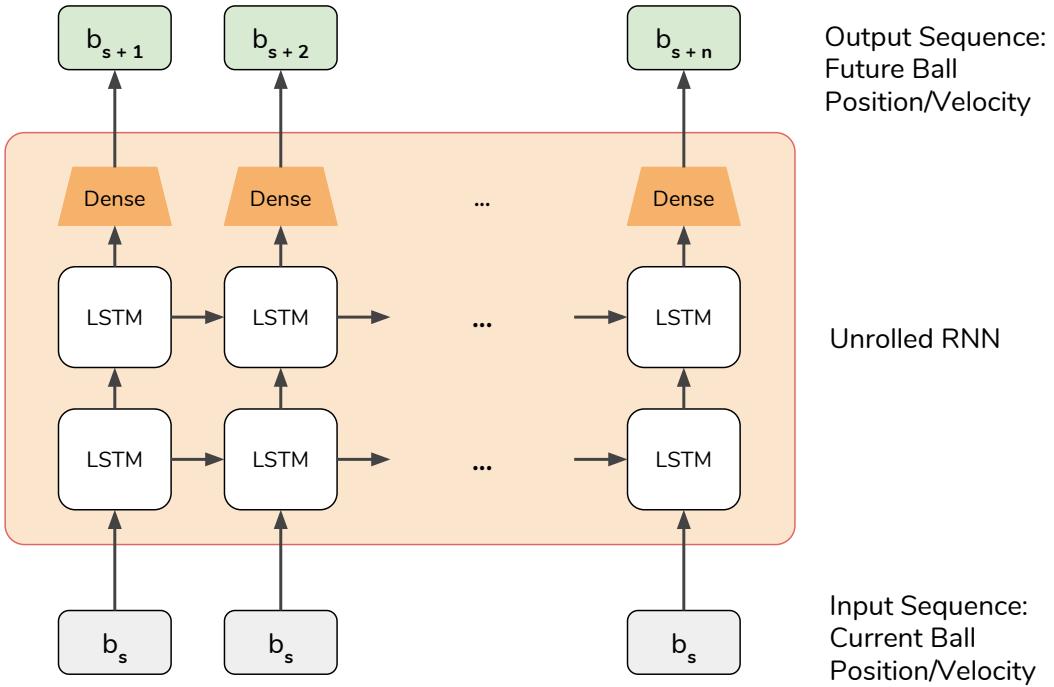
Figure 6.1: **Ball-Trajectory Prediction Model.** Given an estimate on the current position and velocity of the ball, predicts the future position and velocity of the ball over multiple timesteps. The recurrent model has two LSTM layers followed by a fully-connected layer. The figure shows the model unrolled through time for better visualization. At all timesteps, the model receives the same input b_s, which represents the current ball position and velocity.

a candidate paddle-motion state at time of contact p_t, the landing-prediction model predicts where and with what speed the ball will land on the table if the paddle-motion state is achieved.

The landing-prediction model is described by the function

$$\hat{g} = L(p_t, b_t), \tag{6.2}$$

where L denotes the landing model, and $\hat{g}$ denotes the ball's predicted position and speed at landing time.
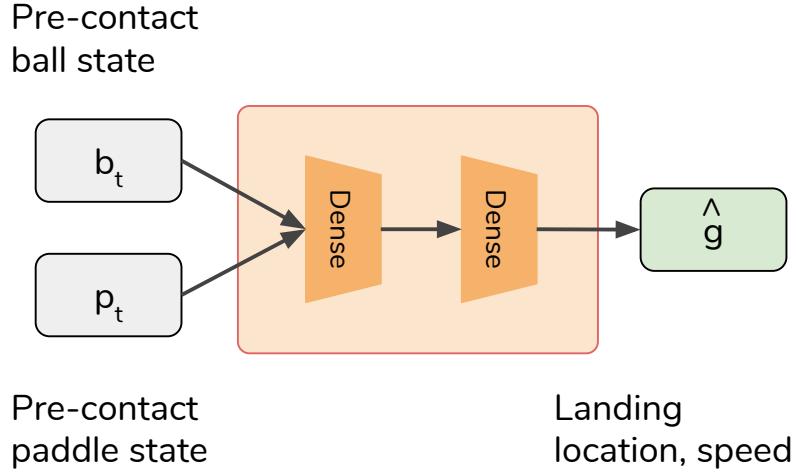
Figure 6.2: **Forward Landing-Prediction Model.** It is a feed-forward network with two fully-connected hidden layers. Given pre-contact ball-motion and paddle-motion states, the model predicts the eventual landing location and speed of the ball when it hits the opponent side. Such a prediction can inform the agent about the outcome of available actions.

The model's architecture is shown in Fig. 6.2. It is a feed-forward network, which produces its output in one step. Since the land-ball policy is only concerned with the eventual landing location and speed of the ball, a feed-forward is preferred since it is faster at inference time and easier to train.

The training data for this model can be limited to the ball trajectories that actually hit the table after contact with the paddle. Alternatively, the notion of landing location can be extended to include positions off the table. In that case, the landing location is picked to be the last location for the ball before it falls below the plane of the table's surface.

The landing-prediction model is used by the land-ball skill to search in the space of candidate paddle actions and select one that is more likely to achieve the desired landing target. In other words, given a set of potential paddle actions p_k, the land-ball policy

selects the action whose predicted landing location and speed is closest to the requested target g.

6.2.3 Inverse Landing-Prediction Model

The search process for selecting paddle actions is computationally expensive. The inverse landing-prediction model addresses this issue by directly predicting the paddle action given a desired landing target.

The landing-prediction model is a forward model, as it predicts the future state of the environment given observations from the current state and an action. It is possible to train an inverse landing model from the same data that is used to train the forward landing-prediction model. The inverse landing model is described by the function

$$p_t = L^{-1}(b_t, g) \tag{6.3}$$

where L^{-1} denotes the inverse landing model.

The model's architecture is shown in Fig. 6.3. Given a pre-contact ball-motion state and a desired landing location and speed, the inverse landing model predicts the pre-contact paddle-motion state that would achieve the desired landing location and speed. The predicted paddle-motion state can be used as an action for the paddle-control skill.

Ignoring the noise in the environment and imperfections in paddle-control, the forward landing-prediction model should have a single answer. That is, if the pre-contact ball and paddle-motion states are accurately known, there is only one possible outcome for the landing location and speed of the ball. The same statement does not hold for the inverse landing model. In general, there might be more than one way to hit a given ball back toward a given target. However, it is still useful to build an inverse landing model by training a neural network on all observed trajectories from human data. Such a model would predict the mean of all observed paddle actions taken by humans. Due to the nonlinearities in the action dimensions, it is possible that the mean action would not be a good solution to the inverse landing problem. However, it can serve as a starting point for a search using the forward landing model.

The inverse landing model is more complex than the forward landing model, since the paddle-motion state p_t has more dimensions than the landing target g. In the current
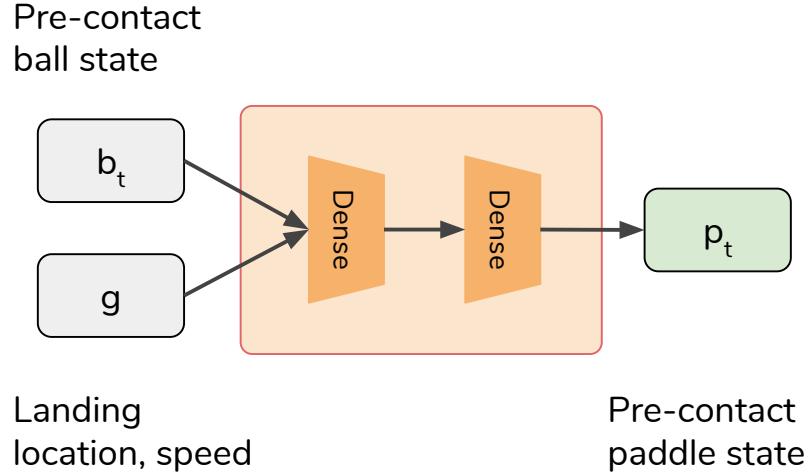
Figure 6.3: **Inverse Landing-Prediction Model.** It is a feed-forward network with two fully-connected hidden layers. Given pre-contact ball-motion state and a desired landing target, the model predicts the paddle-motion state right before contact. The predicted paddle-motion can be used as an action for the paddle-control skill.

implementation, p_t is 12-dimensional, while g has up to four dimensions. Note that some dimensions do not need to predicted. For example, the paddle position can be decided directly based on the predicted position of the ball at the moment of contact, and the height of the landing target is equal to the height of the table plus the radius of the ball.

6.3 Domain Invariances and Data Normalization

This section describes the normalizing transformations that are applied to the collected training data when training the dynamics models discussed in Sec. 6.2.

6.3.1 Invariances in Table Tennis

Normalizing the data reduces the dimensionality of the inputs and outputs to the models and improves sample efficiency. Consider a trajectory containing a number of observations on the position and velocities of the ball and the paddle. In the table-tennis domain, the following invariances hold:

- **Translation invariance across** x, y: Shifting the x, y coordinates of the ball and paddle positions by a constant amount for all the points in the trajectory produces a valid trajectory. This transformation does not affect the object orientations or velocities.

- **Rotation invariance around** z: Rotating all object poses and velocity vectors around any line orthogonal to the $x - y$ plane produces a valid trajectory.

- **Rotation invariance around paddle normal** N: Rotating the paddle poses around the surface normals produces a valid trajectory. The force exerted by the paddle on the ball depends on the contact angle between the ball and their velocities before contact, but it is not affected by the placement of the paddle handle. Rotating the paddle handle around the paddle surface normal does not change the contact dynamics.

- **Inversion invariance for paddle normal** N: Inverting all three elements of the paddle normals produces a valid trajectory. Flipping the paddle normal amounts to changing a forehand to a backhand. As long as the paddle position and velocity stay the same, a forehand and backhand contact have the same impact on the ball.

These variances can be exploited when training the dynamics models from observation trajectories in two ways:

1. Data Augmentation: For any collected trajectory, random perturbations based on the explained invariances can be applied to all points consistently to generate augmented training trajectories.

2. Data Normalization: The collected trajectories can be normalized to remove the redundant dimensions.

Data augmentation has the advantage that it has has a simple implementation; it just results in generating more data. The clients that query the dynamics models do not need to be modified. Another advantage to data augmentation is that it can factor in the position of the table and its impact on the trajectory. For example, a ball that bounces once on the table may not hit the table at all if its x and y coordinates are shifted too much. Similarly, if an augmented trajectory hits the net, it can be removed.

A disadvantage of data augmentation is that it introduces additional hyperparameters. What translation and rotation values are likely to generate trajectories that are valid and likely to occur during the game? How many augmented trajectories should be generated from each recorded trajectory to capture different types of transformations over object states sufficiently? Lastly, the expected accuracy of the data augmentation approach is upper bounded by the data normalization approach.

The data normalization approach does not add any new trajectories. Instead, it just modifies the collected trajectories to remove the redundant dimensions. For example, all ball trajectories can be shifted so that the initial x, y coordinates of the ball are at $(0, 0)$. It has the advantage that it does not introduce any hyperparameters and does not increase the size of the training dataset. Also, reducing the number of dimensions simplifies the problem for the neural network, whereas with data augmentation the neural network needs to capture the invariances in its hidden layers. The disadvantage of normalization is that it cannot model the table. For example, normalizing the trajectory of a bounced ball assumes implicitly that the ball will bounce in any direction and will never hit the net. Lastly, querying a model trained on normalized data requires normalizing the inputs and un-normalizing the outputs. Therefore, it complicates the implementation.

The method implemented in this dissertation uses data normalization. Since the models are not aware of the location of the table and the net, it is up to the higher-level skills like strategy to pick targets that are feasible and increase the agent's reward. In the case of the ball-trajectory prediction model, it is harmless to assume that the opponent's ball will always bounce on the player's table. The agent can plan to respond to a bouncing ball. If that does not happen, the agent simply wins a the point. For the landing model, the strategy skill is expected to pick targets such that the actions recommended by the models are effective. For example, if the strategy skill picks a landing target close to the net, it should pick a lower landing velocity to avoid hitting the net. Therefore, the dynamics models do not need to be aware of the location of the table and the net.

The following section explains the normalizing transformations that are applied to the data used for training each dynamics model.

6.3.2 Normalizing Ball Trajectories

Ball trajectories are normalized as follows:

1. All ball-motion states $\{b_0, \ldots, b_n\}$ are shifted such that the x, y coordinates of the first ball-motion state b_0 become zero. Suppose the position of the first ball b_0 in the original trajectory is

$$l(b_0) = l_x(b_0), l_y(b_0), l_z(b_0), \tag{6.4}$$

where l_x, l_y, l_z denote the x, y, z coordinates of the ball's position. Then all points b_i in the original trajectory are transformed to points b_i' in the normalized trajectory as

$$l(b_i') \leftarrow l(b_i) - (l_x(b_0), l_y(b_0), 0). \tag{6.5}$$

In particular, the first point is transformed to b_0' such that

$$l(b_0') = 0, 0, l_z(b_0). \tag{6.6}$$

This transformation does not affect the ball velocity vectors.

2. All ball positions and velocity vectors are rotated such that the y component of the velocity vector of the first ball b_0 becomes zero. More specifically, the objects and their velocity vectors are rotated around the vertical line

$$x = l_x(b_0), y = l_y(b_0) \tag{6.7}$$

by an an angle ψ equal to

$$\psi = \angle(\mathrm{proj}_{z=l_z(b_0)} v(b_0), (1, 0, 0)), \tag{6.8}$$

where $\angle$ denotes the angle between the two enclosed vectors, $\mathrm{proj}_{z=l_z(b_0)}$ denotes the projection of the velocity vector $v(b_0)$ onto the horizontal plane specified by $z = l_z(b_0)$, and $(1, 0, 0)$ is the unit vector parallel to the x axis.

These transformations remove three of the six dimensions in the input to the ball-trajectory prediction model. Therefore, they simplify the job of the neural network that is modeling ball trajectories.

6.3.3 Normalizing Landing Trajectories

Landing trajectories are normalized as follows:

1. All ball-motion states $\{b_0, \ldots, b_n\}$ and paddle-motion states $\{p_0, \ldots, p_n\}$ are shifted such that the x, y coordinates of the pre-contact ball p_t become zero. Suppose the position of the pre-contact ball b_t in the original trajectory is

$$l(b_t) = l_x(b_t), l_y(b_t), l_z(b_t). \tag{6.9}$$

Then all ball-motion states b_i and paddle-motion states p_i in the original trajectory are transformed in the normalized trajectory as

$$l(b_i') \leftarrow l(b_i) - (l_x(b_0), l_y(b_0), 0), \tag{6.10}$$
$$l(p_i') \leftarrow l(p_i) - (l_x(b_0), l_y(b_0), 0). \tag{6.11}$$

This transformation does not affect any velocity vectors or paddle orientations.

2. All ball and paddle poses and velocity vectors are rotated such that the y component of the velocity vector of the pre-contact ball b_t becomes zero. More specifically, the objects and their velocity vectors are rotated around the vertical line

$$x = l_x(b_t), y = l_y(b_t) \tag{6.12}$$

by an angle ψ equal to

$$\psi = \angle(\text{proj}_{z=l_z(b_t)} v(b_t), (1, 0, 0)). \tag{6.13}$$

59

3. All paddle orientations are replaced by paddle normals.

4. All paddle normals with negative x components are inverted. In other words, all backhand paddles are replaced with forehand paddles.

The first two transformations above remove three of the six dimensions from the ball-motion state input to the landing-prediction model. The third transformation removes one of the 13 dimensions from the paddle-motion state input to the model. The last transformation cuts the space of three of the paddle-motion state dimensions in half. Therefore, normalizing the landing trajectories makes it easier for the neural network to predict landing targets.

It is useful to note that the trajectories are not invariant to translation across z. Changing the height of the ball and paddle invalidates the trajectory if the ball contacts the table at any point. On the other hand, for the landing-prediction model, data augmentation is used to generate augmented trajectories with *reduced* ball and paddle heights. The landing model is not concerned with the motion of the ball after its contact with the table. Given a ball trajectory that collides with the table at the end, it is possible to compute where the collision point would be if the ball was shot from a lower height. The same does not hold for increasing the height of the ball. This property is used to generate multiple landing trajectories with lower heights from each original landing trajectory.

6.4 Learning Dynamics from Demonstrations

The data for training the dynamics models is collected in a VR environment that is integrated with the simulation environment. The VR environment is specially-designed for the purpose of data collection only. The paddle-strike trajectories and ball-motion data collected from human demonstrations

6.4.1 Data Collection in VR Environment

Fig. 6.4 shows the data collection process in the VR environment. A player is controlling the simulated paddle by moving the real VR controller in their hand. The player returns the balls coming from a ball launcher on the other side of the table. The paddle and ball trajectories are recorded and used in training the dynamics models.
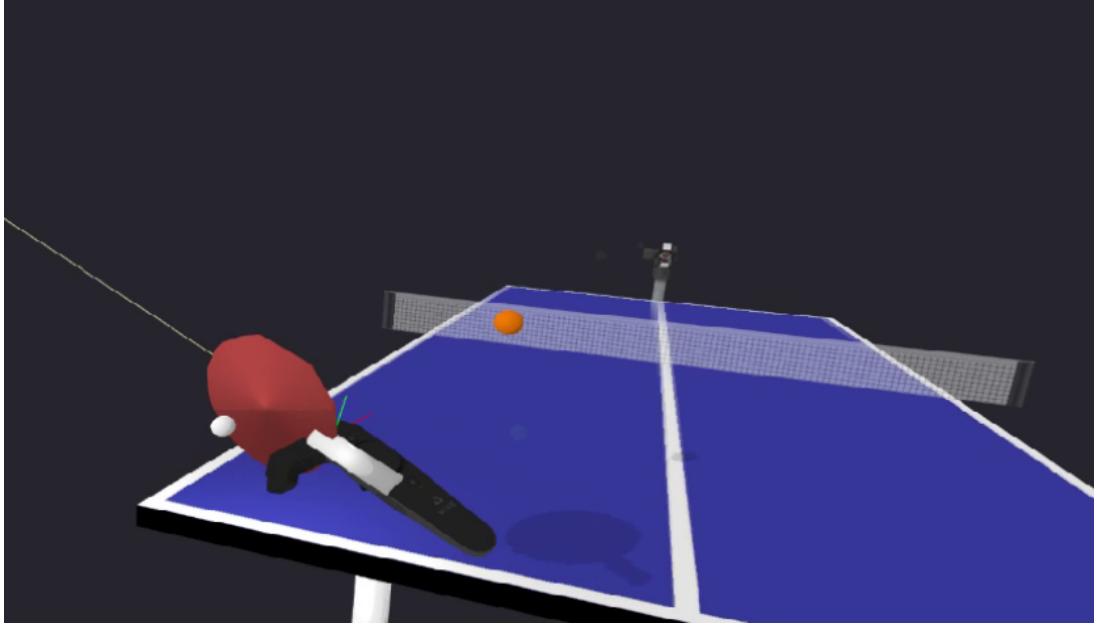
Figure 6.4: **Data Collection in VR Environment.** The VR environment allows a human player to control a simulated paddle by moving the VR controller in the real world. The simulated paddle follows the VR controller (visualized as the black object holding the paddle). The player returns the balls coming from a ball launcher on the other side of the table. The paddle and ball trajectories are recorded and used in training the dynamics models.

Since the VR hardware has only one headset and does not support two players, the data collection environment emulates a two-player game by having the ball launcher match the direction and velocity of the balls that are successfully returned by the player. If the player lands the ball on the other side, the simulator sends the next ball to the corresponding location on the player's side of the table, making the player respond to their own shot. This setup allows the distribution of ball trajectories to be closer to what might be observed in a real two-player game. If the player sends the ball out, the next ball is shot from the ball launcher.

Once the data is collected, it is used to extract ball-motion trajectories and landing trajectories. Ball motion trajectories start at the moment when the ball launcher throws the ball and include the motion state of the ball in every subsequent timestep. Landing trajectories start a few timesteps before contact between the ball and paddle happens

and continue until the ball lands on the opponent side, or crosses the plane at the surface of the table. The ball-trajectory prediction model is trained on the ball trajectories and the landing-prediction model is trained on the landing trajectories. For training the landing-prediction model, only two timesteps of the trajectory are used: a timestep before contact happens, and the final timestep when the ball has landed or has gone out. Since the dynamics models are normalized, a ball that goes out contains useful information as well, since the same paddle motion can be useful for landing a similar ball from a different region of the table.

6.4.2   Data Augmentation

The data collected from human demonstrations contained about 300 successful paddle strikes where the human player was able to make contact with the ball. Strikes where the ball hit the edge of the paddle were removed from the dataset, since that type of contact is not a behavior from which the agent is expected to learn. To speed up data collection, a data augmentation process was used to generate more samples from the 300 human demonstrations. During data augmentation, the paddle and ball trajectories observed during demonstrations were replayed in the simulator with small amounts of noise to produce additional samples. Each additional sample created this way is counted as an extra sample.

6.4.3   Subsampling

A subsampling process is used to extract multiple training samples from each trajectory. Since the sensors have a frequency higher than the environment, each trajectory can be subsampled with multiple offsets. For example, in a landing trajectory, there are multiple sensory observations of the ball and paddle in the 20 milliseconds (one environment timestep) leading up to contact between the ball and paddle. Any one of those observations can be used for training the forward and inverse landing-prediction models. So, with the subsampling process, every landing trajectory can be used to produce 20 samples.

In addition, the ball trajectories are replicated by considering any of the observations in the first ten timesteps as the starting point for the remainder of the trajectory. The training samples extracted with subsampling were not counted as additional samples.

Although not attempted in the current implementation, there is also an opportunity to extract additional landing trajectories by considering a height reduction and computing where the landing location would have been if the height of the paddle and the ball at the moment of contact were reduced. The new landing location can be computed as the intersection of the observed trajectory and an imaginary plane of where the table would be given the height adjustment. This same process can not be used for augmenting free-moving ball trajectories, as the ball-trajectory prediction model is expected to predict the behavior of the ball after it bounces off the table as well, and that behavior can not be computed since reducing the initial height of the ball changes the contact parameters for when it hits the table. The height reduction technique was not used in the current implementation.

6.5 Evaluation

To better evaluate the impact of the number of samples on the accuracy of the dynamics models, they are trained with two dataset sizes. The smaller dataset contains about 7 000 successful paddle strikes where the human player was able to make contact with the ball. The larger dataset contains 140 000 strikes. Once trained, the dynamics models are evaluated on 1000 strikes generated against a ball launcher. These strikes are not part of the training or evaluation datasets. The observed ball and paddle trajectories resulting from the 1000 strikes are recorded and compared against the predictions from the dynamics models. The following sections report the mean errors for the ball-trajectory and landing-prediction models.

6.5.1 Ball-Trajectory Prediction

Fig. 6.5 and Fig. 6.6 show the average position and velocity errors in predicted ball trajectories. Each evaluated ball trajectory contains 30 timesteps of observations corresponding to 0.6 seconds of data. This amount of time is usually enough for the ball to cross the table and reach the striking player. As the plots show, the average position error stays less than 1 cm after 25 timesteps, and the average velocity error stays less than 0.1 m/s after 25 timesteps.

There is little difference between the accuracy of the model trained on the large dataset and the model trained on the small dataset. With data normalization, the number

of inputs to the ball-trajectory prediction model is reduced to three. Moreover, the subsampling process generates many more samples from each recorded ball trajectory. In addition, the physical laws governing the behavior of a bouncing ball are relatively simple to learn. Therefore, it seems this model does not need many samples to learn to predict the behavior of the ball.
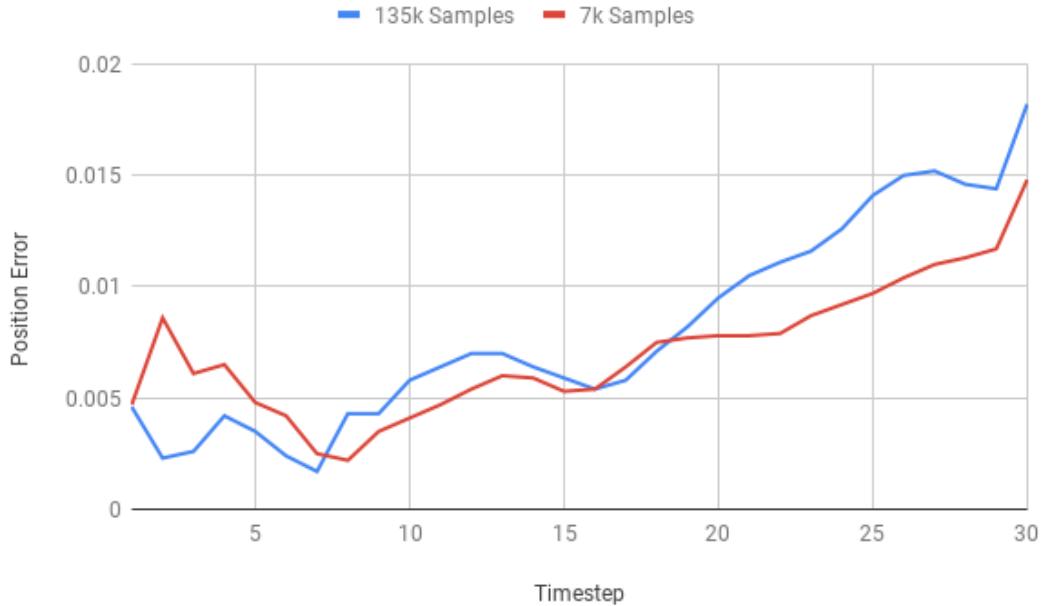
Figure 6.5: **Mean Position Error in Ball-Trajectory Predictions.** The plot shows the mean position error over 1000 ball trajectories containing 30 timesteps of observations each. The error reported is the Euclidean distance between the predicted position and the observed position of the ball. The average position error stays less than 1 cm after 25 timesteps, which suggests that the approach is effective.

6.5.2 Landing Prediction

Table 6.1 shows the mean position error over 1000 landing predictions from models trained on the small and large datasets. The error is about 0.19 m when the model is trained on 7 000 samples and about 0.114 m when the model is trained on 140 000 samples. The landing-prediction model is more complex than the ball-trajectory prediction models,
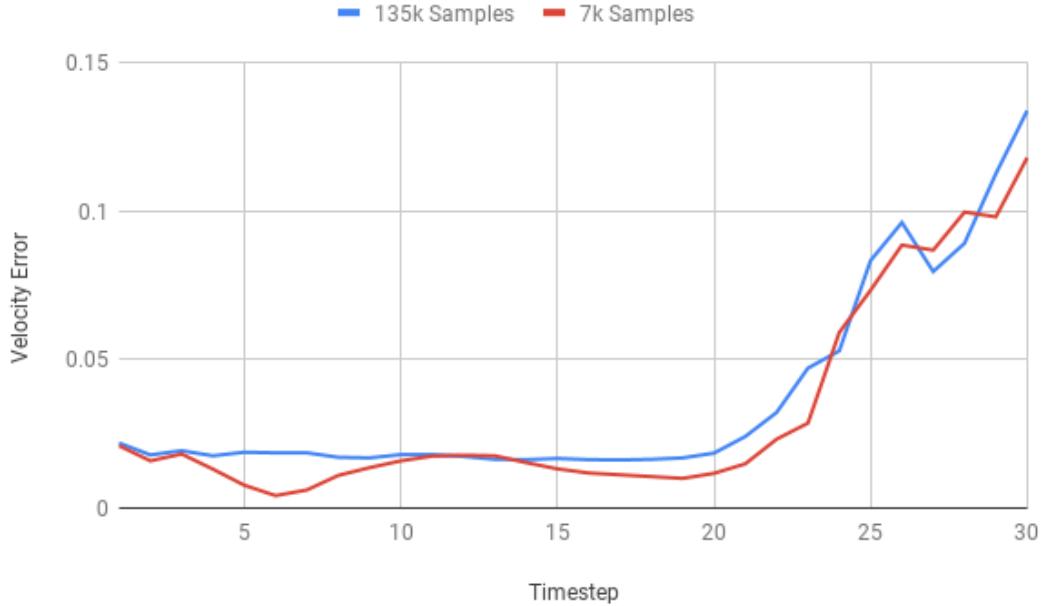
Figure 6.6: **Mean Velocity Error in Ball-Trajectory Predictions.** The plot shows the mean velocity error over 1000 ball trajectories containing 30 timesteps of observations each. The error reported is the Euclidean distance between the predicted 3D velocity and observed 3D velocity vectors for the ball. The average velocity error stays around 0.02 m/s for the first 20 timesteps and starts climbing from there. The 20th timestep is around the time when the ball hits the table. It is likely that predicting the behavior of the ball after contact is more challenging than predicting its free motion for the model. At any rate, the prediction error on velocity remains low compared to the magnitude of observations (around 6 m/s), which suggests that the approach is effective.

since its inputs include both the ball and paddle states. Moreover, the model is expected to predict the eventual position of the ball after it has travelled for 1-2 meters. These reasons might be why the landing-prediction model benefits from more training data.

6.6 Conclusion

The dynamics models discussed in this chapter are only concerned with the physics of the table-tennis environment. They do not deal with the physics of the robot. There-

Samples	Mean Position Error
7 000	0.190 m
140 000	0.114 m

Table 6.1: **Mean Position Error for the Landing-Prediction Model.** Mean position error for models trained from 7 000 trajectories and 140 000 trajectories.

fore, the models can be trained from data collected from human games or practice sessions against a ball launcher. The evaluations show that models are able to predict the motion of the ball over multiple timesteps in the future. There is no noise in the simulation, which makes the simulation outcomes deterministic and easier to predict. Yet, the experiment results show that the models have the ability to capture the physics of interactions between the objects in the environment. Chapter 10 describes an extension to the method that can handle observation noise as well. The next chapter describes the analytic paddle controller that can be used to execute target paddle-motion states to land a given ball at a desired target.

Chapter 7

Paddle-Control Policy

This chapter discusses the implementation of the paddle-control skill. Sec. 7.1 revisits the definition of the paddle-control task and defines some variables used in the rest of the chapter. Sec. 7.2 describes an analytic paddle controller, which is derived mathematically based on the kinematics of the robot links and the motion constraints on the robot motors. Sec. 7.3 describes an analytic paddle-dynamics model that allows higher-level skills to make predictions about expected paddle-motion states resulting from executing the high-level paddle-motion targets with the paddle-control skill. Lastly, Sec. 7.4 discusses an alternative implementation for the paddle-control skill that uses learning.

To increase sample-efficiency, the method in this dissertation uses the analytic paddle controller and does not rely on training to learn the internal dynamics of the robot. The alternative controller that uses learning is studied in an ablation experiment in Chapter 8.

7.1 Paddle-Control Problem

As described in Sec. 4.5, the objective of paddle-control skill is bring the paddle from its current motion-state p_s at time s to the desired paddle-motion target p_t at time t. The target requested from the paddle skill includes the paddle pose $x(p_t)$ and its time derivative $\dot{x}(p_t)$:

$$p_t = x(p_t), \dot{x}(p_t). \tag{7.1}$$

The paddle pose in turn includes the paddle position $l(p_t)$ and surface normal $N(p_t)$:

$$x(p_t) = l(p_t), N(p_t). \tag{7.2}$$

Note that in this formulation, p_t does not fully specify the paddle pose, as there are generally many possible paddle orientations that satisfy the specified paddle normal $N(p_t)$. Specifying the paddle pose with its surface normal instead of a fully-specified orientation has the advantage that it gives the paddle skill the freedom to choose any orientation that satisfies the normal. Also, it is easier to replicate paddle normals from human demonstrations than paddle orientations. The forces exerted from the paddle on the ball at the moment of contact depend on the paddle's normal and stay fixed if the paddle is rotated around its surface normal vector. So any possible orientation that satisfies the given normal will have the same impact on the ball.

The time derivative of the paddle pose $\dot{x}(p_t)$ includes the paddle's linear velocity $v(p_t)$ and angular velocity $\omega(p_t)$:

$$\dot{x}(p_t) = v(p_t), \omega(p_t). \tag{7.3}$$

The paddle's linear and angular velocity at the time of contact affect the forces exerted on the robot and affects the ball's trajectory after contact.

7.2   Analytic Paddle-Control

The analytic paddle controller uses 3D geometry, inverse kinematics, the robot Jacobian, and Reflexxes trajectory planning to achieve the paddle-motion target p_t. It works through the following steps to move the paddle from its current state p_s at time s to a desired state p_t at time t:

1. Find a paddle orientation $r(p_t)$ that satisfies the desired paddle normal in $N(p_t)$.

2. Map the target paddle pose $x(p_t)$ to target joint positions q_t.

3. Map the target paddle velocity $\dot{x}(p_t)$ to target joint velocities $\dot{q}_t$.

4. Compute a joint-trajectory starting with current positions and velocities $q_s, \dot{q}_s$ and reaching the target joint states exactly at time t.

5. Use robot's controller (e.g. a PID controller) to execute joint commands between times s and t to follow the computed trajectory.

The following sections describe each step in detail.

7.2.1 Mapping Paddle's Normal to Orientation

The analytic controller computes a paddle orientation $r(p_t)$ based on the requested paddle position $l(p_t)$ and surface normal $N(p_t)$. First, it uses inverse kinematics to find an intermediate pose that satisfies only the requested position $l(p_t)$:

$$q_t^l \leftarrow \text{IK}_c(l(p_t)), \tag{7.4}$$

where IK_c denotes the inverse kinematics function starting with canonical robot rest poses, and q_t^l denotes the joint positions corresponding to an intermediate pose that satisfies $l(p_t)$.

In the coordinate system introduced in Sec. 2.1 and shown in Fig. 2.1, the x coordinate of the paddle normal points toward the opponent. A normal with a positive x coordinate specifies a forehand paddle, while a normal with a negative x coordinate specifies a backhand paddle. The IK_c function in Eq. 7.4 runs the IK optimization starting with either a canonical forehand or backhand rest pose for the robot depending the x coordinate of the requested paddle. Inverse kinematics is typically implemented as a local optimization process that iteratively uses the robot Jacobian to reduce the distance between the current pose and the requested pose. So, the solution found by inverse kinematics depends on the initial pose of the robot. Starting the search with a rest pose leads to an answer that is closer to that initial pose, and therefore likely to be well within the robot's reachable space.

Once the intermediate solution q_t^l is found, forward kinematics is used to compute the corresponding paddle pose for this solution:

$$x(p_t^l) \leftarrow \text{FK}(q_t^l), \tag{7.5}$$

where FK denotes the forward kinematics function. Assuming that the requested target paddle position $l(p_t)$ is reachable by the robot, p_t^l should satisfy that position, i.e. , one should have:

$$l(p_t^l) = l(p_t). \tag{7.6}$$

Next, the corresponding paddle normal $N(p_t^l)$ at the intermediate solution is computed. Then a minimum rotation between $N(p_t^l)$ and the target paddle normal is computed as:

$$\phi \leftarrow \angle(N(p_t^l), N(p_t)), \tag{7.7}$$

where ϕ denotes a 3D rotation that can move $N(p_t^l)$ to $N(p_t)$. Applying the rotation ϕ to the intermediate paddle orientation produces the desired paddle orientation:

$$r(p_t) \leftarrow \phi(r(p_t^l)), \tag{7.8}$$

where $r(p_t)$ denotes the desired paddle orientation and $r(p_t^l)$ denotes the paddle orientation corresponding to the intermediate paddle pose.

Due to its construction, $r(p_t)$ is guaranteed to have the desired paddle normal $N(p_t)$. Also, because it is constructed with a minimum rotation from a feasible pose p_t^l, it is likely that $r(p_t)$ is feasible by the robot as well.

7.2.2 Mapping Paddle's Pose to Joint Positions

Inverse kinematics can be be used to find some joint positions q_t to satisfy the paddle pose $x(p_t)$ subject to the physical limits of the robot and the limits on the range of positions for its joints:

$$q_t \leftarrow \text{IK}(x(p_t)), \tag{7.9}$$

where IK denotes the inverse kinematics function. In other words, IK maps the desired paddle pose $x(p_t)$ to a robot pose q_t.

In general, there are multiple solutions to the IK problem. The method in this dissertation uses null-space control to prefer robot poses that are closer to some canonical forehand and backhand poses.

7.2.3 Mapping Paddle's Linear and Angular Velocities to Joint Velocities

To map the desired linear and angular velocities for the paddle $\dot{x}(p_t)$ to some joint velocities, the end-effector Jacobian is computed at pose q_t:

$$J_t \leftarrow \frac{\partial x(p_t)}{\partial q_t}, \tag{7.10}$$

where J_t denotes the Jacobian at q_t. Eq. 7.10 can be rewritten as:

$$\frac{\partial x(p_t)}{\partial q_t} = J_t \tag{7.11}$$

$$\frac{\partial x(p_t)}{\partial t} \frac{\partial t}{\partial q_t} = J_t \tag{7.12}$$

$$\frac{\partial x(p_t)}{\partial t} = J_t \frac{\partial q_t}{\partial t} \tag{7.13}$$

$$\dot{x}(p_t) = J_t \dot{q}_t, \tag{7.14}$$

where $\dot{x}(p_t)$ and $\dot{q}_t$ denote the time derivatives of $x(p_t)$ and q_t. In other words, the Jacobian establishes a relationship between the paddle's linear and angular velocity and the robot's joint velocities.

In order to solve for $\dot{q}_t$ given $\dot{x}(p_t)$, the Jacobian needs to be inverted. To handle non-square Jacobians when the robot assembly has more than six joints, and also to avoid failing on singular matrices, the pseudo-inverse method is employed to invert the matrix:

$$J_t^\dagger \leftarrow \text{pseudo-inverse}(J_t). \tag{7.15}$$

Then, the required joint velocities at target can be obtained as:

$$\dot{q}_t \leftarrow J_t^\dagger \dot{x}(p_t). \tag{7.16}$$

The current joint positions and velocities $q_s, \dot{q}_s$ can be obtained directly by reading them from the robot's controller. Therefore, the paddle-control policy can analytically obtain the inputs it needs to pass to the trajectory planning skill as described in Eq. 4.8:

$$\pi_p(t, p_t \mid p_s) = \pi_t(t, q_t, \dot{q}_t \mid q_s, \dot{q}_s) \tag{7.17}$$

7.2.4 Trajectory Planning

At this point the problem of executing the paddle-motion target p_t is reduced to executing target joint states $q_t, \dot{q}_t$ given the initial joint states $q_s, \dot{q}_s$, as described in the trajectory planning skill in Eq. 4.9. This task is accomplished by employing Reflexxes to compute a joint state trajectory between times s and t. Reflexxes is an analytic algorithm which computes the intermediate joints states between times s and t solely based on a set of motion constraints defined on individual joints:

$$
\begin{aligned}
\{(q_j, \dot{q}_j, \ddot{q}_j) \mid s \leq j \leq t-1\} &\leftarrow \text{Reflexxes}(q_s, \dot{q}_s, t, q_t, \dot{q}_t) \\
&\text{subject to:} \\
q_{\min} &\leq q_j \leq q_{\max}, \\
\dot{q}_{\min} &\leq \dot{q}_j \leq \dot{q}_{\max}, \\
\ddot{q}_{\min} &\leq \ddot{q}_j \leq \ddot{q}_{\max}, \\
\dddot{q}_{\min} &\leq \dddot{q}_j \leq \dddot{q}_{\max}.
\end{aligned}
\tag{7.18}
$$

Reflexxes is able to produce trajectories at the desired control frequency, which is 1 kHz in this implementation. It is a fast library and can plan for the trajectory and return the next step typically within 1 ms.

Reflexxes can compute trajectories that take the minimum time, or trajectories that complete precisely at some specified time t. The method in this dissertation uses both modes for different skills:

1. The positioning skill is active when the agent is awaiting the opponent's action. Its objective is to put the robot in some pose that is suitable for responding to the next incoming ball. So, for this skill, it is desirable to reach the target position as soon as possible. When the positioning skill is active, the paddle skill requests a trajectory that reaches the target in minimum time.

2. The objective of the land-ball skill is to hit the ball back at a planned contact time t. When producing trajectories for this skill, the paddle skill is always given a desired target time t. In such cases, usually the robot starts moving slowly and builds up speed toward the target just at the right time to allow it to achieve the desired joint velocities $\dot{q}_t$ exactly when it is reaches the pose specified by the joint positions q_t.

It is possible that before the robot reaches the target of the positioning skill, the opponent hits the ball back and the land-ball skill becomes active again. In that case, the trajectory planned for the positioning skill is not completed, and the trajectory for the land-ball skill starts with the current joint states as its initial condition.

There are situations where no feasible trajectories exist that satisfy the constraints. For one, the initial joint states at time s or the final joint states at time t might violate the position and velocity constraints of the robot. In the hierarchical setup, this may happen due to the higher-level skills requesting a paddle-motion target that requires executing joint velocities beyond the limits of the robot. In such cases, $\dot{q}_t$ already violates the constraints in Eq. 7.18. Even with conservative limits on the paddle's velocity $\dot{x}(p_t)$, the joint velocities $\dot{q}_t$ may end up being high when the paddle is close to the singularity points of the robot. In such regions, the inverse Jacobian matrix computed in Eq. 7.15 contains elements with large magnitudes. In situations where the requested final joint positions and velocities $q_t, \dot{q}_t$ are invalid, the analytic controller does not send any commands to the robot.

Another class of infeasible trajectories are those with insufficient time. For example, if a higher-level skill demands that the paddle moves from one side of the table to the other side in 0.1 seconds, the trajectory would violate the acceleration and jerk limits of a typical robot. In such cases, Reflexxes can still compute a minimum-time trajectory towards $q_t, \dot{q}_t$. However, due to having insufficient time, at time t the robot will be at some state $\hat{q}_t, \hat{\dot{q}}_t$ somewhere in between the starting state and the target state. This state can be queried and used to evaluate the expected outcome of the action under consideration:

$$\{..., (\hat{q}_t, \hat{\dot{q}}_t, \hat{\ddot{q}}_t)\} \leftarrow \text{Reflexxes}(q_s, \dot{q}_s, t, q_t, \dot{q}_t). \tag{7.19}$$

7.2.5 Joint-Control

Once a trajectory is computed, it needs to be executed on the robot. The joint-trajectory includes joint positions, velocities and accelerations for each timestep. When the control frequency is high enough, the points on the trajectory are so close to each other that just sending the joint positions $q_i, ..., q_t$ to a PID controller can control the robot smoothly. Some robots have PID controllers that also handle velocity targets. For such controllers, the joint velocities $\dot{q}_i, ..., \dot{q}_t$ from the trajectory can also be fed to the robot's controller. Another option for joint-control is inverse dynamics control, where the kinematics and inertial properties of the robot links are used to directly compute the forces or torques for joints. In either case, the underlying controller that is available for the robot implements the policy

$$\begin{aligned} \pi_r(u_j \mid q_j, \dot{q}_j, \ddot{q}_j, q_{j+1}, \dot{q}_{j+1}, \ddot{q}_{j+1}), \\ s \leq j \leq t - 1, \end{aligned} \tag{7.20}$$

where $q_j, \dot{q}_j, \ddot{q}_j$ denote the current joint positions, velocities, and accelerations, $q_{j+1}, \dot{q}_{j+1}, \ddot{q}_{j+1}$ denote the desired joint positions, velocities and accelerations at the next timestep, and u_j denotes the joint-control command to execute in order to achieve the desired joint states at the very next timestep.

7.3 Paddle-Dynamics Model

The objective of the paddle-control skill is to control the robot in a way to achieve the requested paddle-motion target p_t at time t. As outlined in Sec. 7.2, this skill is implemented with an analytic controller. However, the solution found by the analytic controller may not always achieve the desired paddle-motion target p_t due to the following reasons:

1. Failure in inverse kinematics: The desired paddle-motion target p_t may not be physically feasible for the robot. The target paddle position may be out of reach for the robot, or it may have an orientation that is not possible for the robot's anatomy.

2. Failure in trajectory planning: The desired target time t may be too close. In that case, the trajectory planning skill cannot move the robot to the target in time without violating the specified motion constraints.

The analytic controller can predict the error in achieving the paddle-motion target due to either of the above two causes. If there is enough time for the trajectory to reach the target while satisfying the motion constraints, then the trajectory would be feasible. As shown in Eq. 7.19, the final point on the trajectory computed by Reflexxes contains information about where the robot will be at target time t. When the trajectory is feasible by the given time t, the final joint positions and velocities $\hat{q}_t, \dot{\hat{q}}_t$ will be equal to the planned target $q_t, \dot{q}_t$. In other words, the following holds:

$$\hat{q}_t = q_t, \tag{7.21}$$
$$\dot{\hat{q}}_t = \dot{q}_t. \tag{7.22}$$

When the time given is not enough to complete the trajectory, $\hat{q}_t, \dot{\hat{q}}_t$ can be used to predict the final paddle-motion state at time t. First, forward kinematics is used to predict the paddle pose from executing $\hat{q}_t$:

$$x(\hat{p}_t) \leftarrow \text{FK}(\hat{q}_t) \tag{7.23}$$

where $x(\hat{p}_t)$ denotes the predicted resulting paddle pose.

Then, the end-effector Jacobian is used to produce a prediction for the paddle's final linear and angular velocities given the predicted final joint velocities $\dot{\hat{q}}_t$:

$$\dot{x}(\hat{p}_t) \leftarrow J(\hat{q}_t)\dot{\hat{q}}_t. \tag{7.24}$$

Equations 7.23, 7.24 combined specify a prediction for the full paddle-motion state at time t:

$$\hat{p}_t = x(\hat{p}_t), \dot{x}(\hat{p}_t). \tag{7.25}$$

In Sec. 7.2, inverse kinematics (Eq. 7.9), the robot Jacobian (Eq. 7.16), and Reflexxes (Eq. 7.19) were used to map the paddle-motion target p_t to joint positions and velocities $\hat{q}_t, \dot{\hat{q}}_t$. In this section, forward kinematics (Eq. 7.23) and the robot Jacobian (Eq. 7.24) are used to map $\hat{q}_t, \dot{\hat{q}}_t$ to a predicted paddle-motion state $\hat{p}_t$ resulting from running the analytic controller. Combining the above equations permits defining a forward paddle-control model, which given the current paddle-motion state p_s and a paddle-motion target p_t produces a prediction for the paddle-motion state resulting from running the analytic controller:

$$\hat{p}_t \leftarrow P(p_t, p_s) \tag{7.26}$$

where P is the forward paddle-control model under the motion constraints specified in Eq. 7.18.

The prediction $\hat{p}_t$ shows the expected state of the paddle at the planned contact time with the ball. This prediction can be used in conjunction with the forward landing model from Sec. 6.2.2 to inform the agent about the expected landing location and velocity resulting from executing the paddle action p_t.

The controller discussed in Sec. 7.2 and the paddle-control model discussed in this section are entirely analytic. They are derived mathematically based on the kinematics of the robot links and the motion constraints on the robot motors. This approach increases sample efficiency since no training episodes are being spent on learning robot-control. Moreover, the abstract control space exposed by the analytic controller makes the remaining parts of the problem easier to learn with reinforcement learning.

7.3.1 Learning Paddle-Dynamics

One of the advantages of using a trajectory planning module like Reflexxes is that it generates smooth targets which already account for the physical motion limits of the robot. This smoothness and continuity in the targets can hide away some imperfections in the underlying robot controller. For example, if the PID gains are not tuned very well, they would cause smaller errors or oscillations when the targets are closer to each other.

However, robot controllers and robots are ultimately imperfect and imprecise. There are various causes for deviations to exist between the expected behavior and observed behavior of a robot. Such deviations exist due to:

- Imperfections in the controller's implementation, gains, and other parameters.

- Round-trip delay of executing commands on the robot.

- Malfunctioning or worn out robot parts.

- Mechanical backlash caused by gaps between components.

- Misspecified motion constraints. If the velocity, acceleration, and jerk limits given to Reflexxes are higher than the actual limits of the robot, Reflexxes would compute trajectories that are beyond the physical limits of the robot.

It is possible to extend the notion of the robot's model to also capture such imprecisions in control of the robot. Unlike the analytic paddle-control model in Sec. 7.3, which was derived mathematically, it is best to learn a dynamics model over robot's behavior by experimenting and observing outcomes. A neural network can be trained to predict inaccuracies in robot-control regardless of the underlying cause. As the robot is executing the target joint-motion states specified by $q_t, \dot{q}_t$, the resulting joint positions and velocities

at time t can be recorded and used as labels for training a model. The trained model can then make predictions of the form:

$$\hat{\hat{q}}_t, \hat{\dot{\hat{q}}}_t \leftarrow R(q_s, \dot{q}_s, q_t, \dot{q}_t) \tag{7.27}$$

where R denotes the forward robot-control model, and $\hat{\hat{q}}_t, \hat{\dot{\hat{q}}}_t$ denote the expected joint position and velocity observations at time t.

This constitutes a forward prediction, which when combined with Eq. 7.26 can produce a more accurate prediction about the future state of the paddle. Such a forward prediction can be used to produce a more accurate estimate on the landing location of a particular strike specified by p_t.

In the other direction, the same training data can be used to learn a corrective robot-control model as in:

$$q_t{}', \dot{q}_t{}' \leftarrow R^{-1}(q_s, \dot{q}_s, q_t, \dot{q}_t), \tag{7.28}$$

where R^{-1} denotes the inverse robot-control model, and $q_t{}', \dot{q}_t{}'$ are alternative targets such that if they are requested from the robot, it is expected that the observed joint states at time t would be close to the actual targets $q_t, \dot{q}_t$. In other words, it is expected that:

$$q_t, \dot{q}_t \approx R(q_s, \dot{q}_s, q_t{}', \dot{q}_t{}'). \tag{7.29}$$

The inverse robot-control model can be used to adjust the joint position and targets before they are passed to the trajectory planning skill to increase the likelihood that the requested paddle-motion target p_t is going to be achieved.

7.4   Learning Paddle-Control

The primary approach in this dissertation uses the analytic paddle controller discussed in Sec. 7.2. This section discusses an alternative implementation for the paddle-

control skill using learning. The learning approach is undesirable when an analytic solution exists. The learning approach discussed here is implemented and evaluated in an ablation experiment in Chapter 8.

As shown in the task hierarchy from Chapter 4, the paddle-control skill depends on the trajectory planning skill, which in turn depends on the joint-control skill. However, it is possible to treat paddle-control as a single problem. Combining the task specifications for these three skills from Eq. 4.8, Eq. 4.9, and Eq. 4.10 produces a contracted definition for the paddle-control skill as:

$$\pi_p(t, p_t \mid p_s) = \{u_j \mid s \le j \le t - 1\} \tag{7.30}$$
$$\text{subject to motion constraints in Eq. 4.9.}$$

Eq. 7.30 relates the high-level paddle-control policy with p_t as target to the low-level joint-control actions u_j over multiple timesteps from s to $t-1$.

Given this formulation, the paddle-control task can be treated as a learning problem where the objective is to find the right joint commands to bring the paddle to the desired state at the desired time. A basic approach to learning paddle-control with standard reinforcement learning may use random actions on joints with the intention of gradually discovering the right actions that move the paddle toward the requested targets. This approach is not desirable, since it is very inefficient and random joint actions can break the robot.

An alternative approach may consider action spaces that span over time intervals longer than one timestep. Effective strikes generally have some continuity in joint motions. An effective strike usually maintains the direction of motion for most joints. So, one can sample a set of fixed velocity/acceleration profiles for each joint and use those to create paddle strikes. However, it is hard to determine whether such motion profiles would cover the space of all useful strikes. Another problem is the issue of the initial pose of the robot at the start of the strike. If the paddle is already in front of the robot, a successful strike may need to bring the paddle back first, before it can be moved forward again to hit the ball with enough force. Moreover, the robot may be in the middle of some motion and have non-zero velocities on some joints when a new strike is to be executed. These requirements greatly increases the space of possible motions that need to be tried in order to develop a high-level paddle controller via training.

7.5 Positioning Policy

The positioning skill is active when the agent is awaiting the opponent's action, i. e. , when the opponent is striking the ball. This skill is invoked either if the episode starts with the launcher sending the ball toward the opponent, or right after the agent hits the ball back toward the opponent. The skill stays active until the opponent makes contact with the ball, at which point the striking skill becomes active.

The positioning skill receives a paddle position target $l(p)$ from the strategy skill. The objective of the skill is to move the paddle to the requested position as quickly as possible. Note that the requested paddle target p has no specified time.

The paddle position is a proxy to the robot's pose. Specifying the paddle pose instead of the robot pose has the advantage that it makes the policy less dependent on the specific robot that is in use. Moreover, a paddle position can be specified with three values, while the full robot pose typically requires specifying six or more joint positions.

As discussed in Sec. 4.4, the positioning skill π_w is defined by the policy:

$$\pi_w(l(p), \operatorname{sgn} N_x(p)) = \pi_p(p \mid p_s), \tag{7.31}$$

where π_p denotes the paddle-control policy, p denotes some paddle-motion target that satisfies the requested paddle position $l(p)$ and normal direction indicated by $\operatorname{sgn} N_x(p)$. The paddle skill is expected to achieve paddle-motion target p as fast as possible.

The positioning skill is implemented analytically. It simply computes a fully-specified paddle-motion target p which satisfies the paddle position $l(p)$ and forehand/backhand pose indicated by $\operatorname{sgn} N_x(p)$ and requests p from the paddle-control skill, which in turn executes it using the trajectory planning skill. To compute p, the positioning skill first uses inverse kinematics to compute a robot pose as in:

$$q^w \leftarrow \operatorname{IK}_c(l(p)), \tag{7.32}$$

where IK_c denotes the inverse kinematics function starting with some canonical robot rest pose, and q^w denotes the joint positions that satisfy the requested paddle position $l(p)$.

Satisfying sgn $N_x(p)$ is achieved by starting the IK search with a canonical pose which has the same forehand or backhand direction as specified by sgn $N_x(p)$. Since IK algorithms perform a local search, a carefully-chosen canonical forehand and backhand pose allow the positioning policy to satisfy $l(p)$ without flipping the forehand/backhand orientation. Once target joint positions q^w are computed, forward kinematics is used to compute a paddle pose from them:

$$x(p) \leftarrow \mathrm{FK}(q^w). \tag{7.33}$$

Assuming that the requested target paddle position $l(p)$ is reachable by the robot, the paddle pose $x(p)$ should satisfy that position. Otherwise, $x(p)$ will get as close as possible to $l(p)$. The fully-specified paddle-motion target should also include the paddle's linear and angular velocities. In the current implementation, the positioning skill requests a stationary target by setting the target velocities to zero:

$$p \leftarrow x(p), 0. \tag{7.34}$$

However, a more complex implementation could request to have non-zero joint velocities at target to reduce the expected reaction time to the next ball coming from the opponent. The full paddle-motion target p is then sent to the paddle-control policy to execute.

7.6 Conclusion

This chapter discussed the analytic controller setup that handles paddle-control, trajectory planning, and joint-control tasks. It also discussed the analytic paddle-dynamics model P which predicts the expected paddle-motion state $\hat{p}_t$ resulting from executing a desired paddle-motion target p_t. The next chapter explains how the analytic paddle controller and model can be used in conjunction with the dynamics models trained from human data over the environment's game space to implement the land-ball policy.

Chapter 8

Striking Policies

The dynamics models trained from human demonstrations allow the agent to predict the trajectory of an incoming ball, and predict the landing locations resulting from hitting the incoming ball with given paddle strikes. This chapter describes how the model-based land-ball policy uses these dynamics models and the paddle-control policy described in the previous chapter to execute arbitrary landing targets. The land-ball policy is evaluated on a target practice task with random landing targets. In order to determine whether using arbitrary strikes from human demonstrations imposes a restriction on the land-ball policy, the policy is also evaluated on dynamics models trained from data generated directly on the robot. To evaluate the sample-efficiency of the model-based land-ball policy, the land-ball task is also learned directly using a model-free reinforcement algorithm. Lastly, the alternative hit-ball striking policy is described. The hit-ball policy does not use the strikes demonstrated by the humans and is suitable for learning new striking motions.

8.1   Model-Based Land-Ball Policy

The objective of the land-ball skill is to execute a paddle strike to send an incoming ball with motion state p_s to a landing target g consisting of a target position and speed at the moment the ball lands on the opponent's side of the table. This section describes two implementations for the land-ball policy and evaluates both.

8.1.1   Policy Implementation

Fig. 8.1 illustrates the implementation of the model-based land-ball policy using four dynamics models: ball-trajectory prediction, forward landing-prediction, inverse landing-prediction, and the analytic paddle-control model. Algorithm 1 outlines the policy steps in detail.
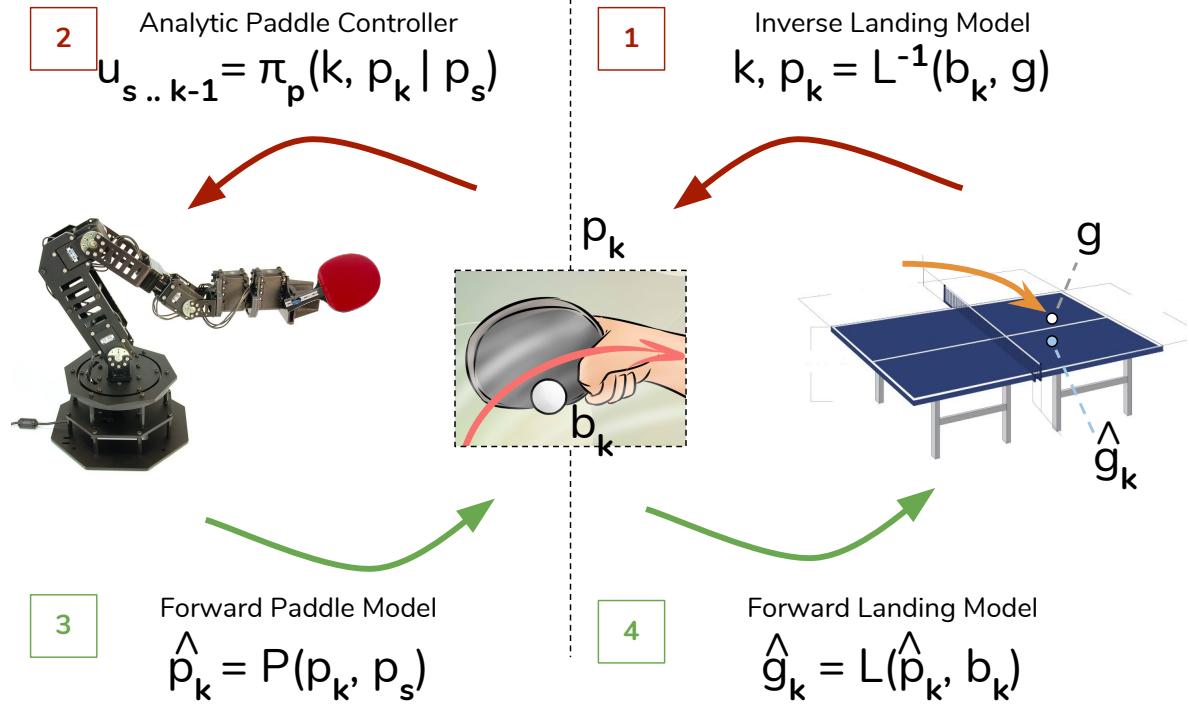
Figure 8.1: **Model-Based Land-Ball Policy.** The land-ball policy uses three dynamics models learned in the game space and an analytic paddle model derived in the robot space to execute land-ball targets specifying a desired position and speed for the ball at the moment of landing. The policy uses the ball-trajectory prediction model (not visualized) to predict the ball's trajectory. For any candidate pre-contact point in the trajectory and the corresponding ball-motion state b_k, it uses the inverse landing model to compute a pre-contact paddle-motion target p_k. For each p_k, the analytic paddle controller computes a joint-trajectory and joint actions to achieve the target p_k, as well as a prediction $\hat{p}_k$ on the expected paddle-motion state resulting from executing the target p_k. Given $\hat{p}_k$ and b_k, the forward landing model can predict $\hat{g}_k$, the landing position and speed resulting from executing p_k. The dynamics models permit implementing the land-ball policy with dynamics models trained from human demonstrations. This policy can be deployed on any robot assembly to hit land-ball targets without prior training.

Fig. 8.2 demonstrates the different stages of the land-ball policy's algorithm. Given the incoming ball's motion state p_s, the policy predicts the ball's future trajectory T. The predicted ball trajectory contains future ball position and velocity observations at the res-

input : Current ball-motion state b_s
input : Desired landing target g
$T = b_{s+1}, b_{s+2}, \ldots, b_n \leftarrow B(b_s)$
foreach $b_k \in T$ *such that* b_k *is reachable* **do**
$\quad | \quad p_k \leftarrow L^{-1}(b_k, g)$
$\quad | \quad \hat{p}_k \leftarrow P(p_k, p_s)$
$\quad | \quad \hat{g}_k \leftarrow L(\hat{p}_k, b_k)$
end
$t \leftarrow \arg\min_k \|\hat{g}_k - g\|$
$i \leftarrow s$
repeat
$\quad | \quad$ emit next action u_i from $\pi_p(t, p_t \mid p_s)$
$\quad | \quad i \leftarrow i + 1$
until *robot paddle hits the ball or episode ends*

Algorithm 1: Model-Based Land-Ball Policy Algorithm

olution of the environment timestep (20 ms). There are multiple options for selecting a subset of the predicted trajectory T as potential striking targets. In the current implementation, a heuristic is used to select all points that lie between the two planes $x = -1.8$ m, $x = -1.6$ m, corresponding to a 20 cm band in front of the robot assembly. This band typically contains 2-3 predicted observations in the predicted ball trajectory. These points are highlighted as light-green balls in Fig. 8.2. Considering multiple potential striking points allows the land-ball policy to come up with multiple striking solutions and pick one that is most likely to satisfy the requested landing target g. Another potential solution for selecting striking targets from the trajectory is to use a heuristic to prefer balls that are closer to a certain preferred height for the robot. It is also possible to leave this decision entirely up to a higher-level skill like the strategy skill. In other words, the strategy skill could additionally specify the striking target by requesting a desired height or distance from the net for the point of strike.

For each potential striking point b_k corresponding to time k, the policy uses the inverse landing-prediction model to compute a target paddle-motion p_k to land the ball at target g. To ensure contact with the ball, $l(p_k)$ is always set to $l(b_k)$. Since the landing-prediction models are trained from human demonstrations, querying the inverse landing-prediction model amounts to computing the expected mean striking motion as demonstrated by humans when aiming for a target g. For each p_k, the forward paddle

84
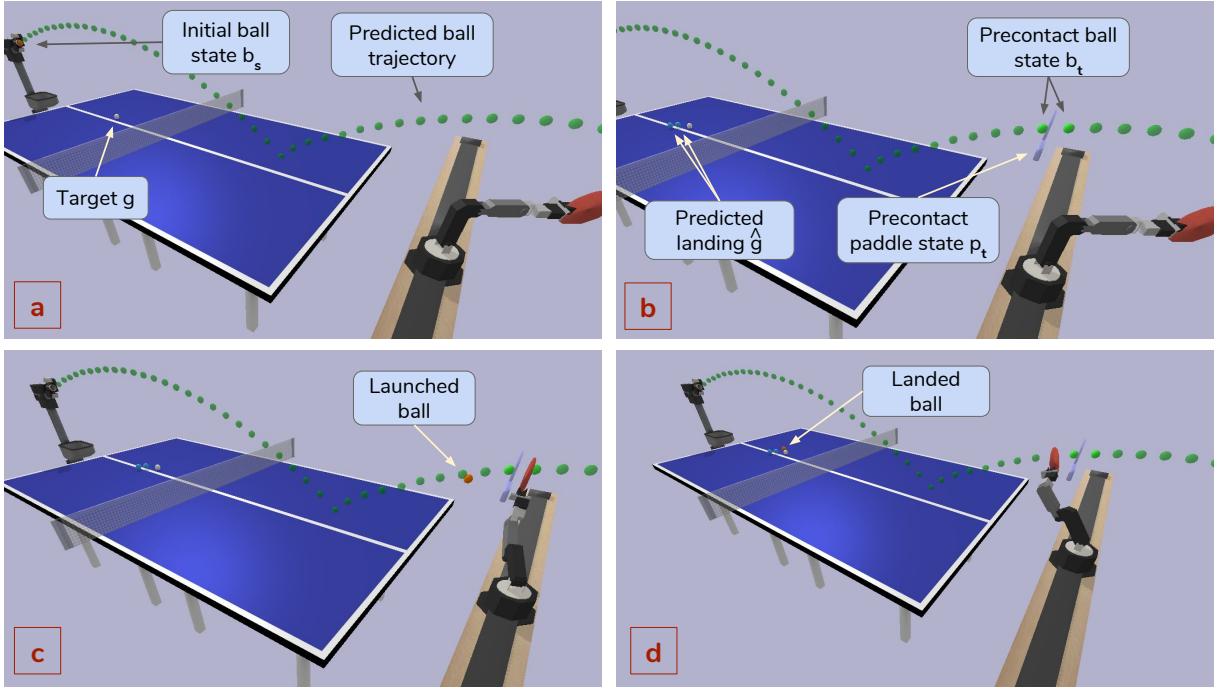
Figure 8.2: **Demonstration of the Land-Ball Policy. a)** A target g is passed as input to the land-ball skill. Given the estimate on the current ball-motion state b_s (visualized by the orange ball), the future ball trajectory (visualized as a sequence of green balls) is predicted. **b)** Two points in the predicted ball trajectory are selected as pre-contact points (visualized by light-green balls). For each such point, a pre-contact paddle-motion target is computed. Only one of the paddle-motion targets is visualized by a transparent paddle. Given each paddle-motion target and its corresponding ball-motion state, the forward landing model is used to predict the landing position and speed of the ball. The paddle solution p_t whose predicted landing $\hat{g}$ is closer to the requested target g is selected. **c)** The paddle-motion target is sent to the analytic paddle controller, which executes it. **d)** The ball lands on the table. Since neither the inverse landing nor the forward landing models are completely accurate, the ball may land at a location that is close to, but not exactly on either the requested target g, or the predicted landing target $\hat{g}$.

model is used to compute a prediction for the expected paddle-motion state $\hat{p}_k$ that would result from executing p_k with the analytic paddle controller. If the pose specified by p_k is not feasible for the robot, or if the target motion state is not reachable in time given the kinematic motion constraints, $\hat{p}_k$ informs the agent about how close the paddle can come

to the requested state p_k. In addition, $\hat{p}_k$ captures any imprecisions resulting from the iterative IK search process. Given each predicted $\hat{p}_k$, the policy uses the forward landing-prediction model to compute $\hat{g}_k$, the expected landing position and velocity resulting from hitting the ball b_k with the paddle strike specified by $\hat{p}_k$. The policy then picks the strike p_k such that it has the smallest predicted landing error and executes it using the analytic paddle-control policy as $\pi_p(t, p_t \mid p_s)$.

8.1.2   Automatic Forehand/Backhand

Any given strike can be executed with forehand or backhand paddle motions. As described in Sec. 6.3, inverting the paddle normal (from forehand to backhand or vice versa) does not change its impact on the ball. The normalization process used in training the dynamics models replaces all backhand strikes from human demonstrations with forehand strikes. So, the inverse landing-prediction model always suggests forehand paddle strikes. However, for some targets backhand strikes might be better choices. For example, if the predicted striking point b_k is close to the left side (from the robot's perspective) of the robot's reachable space, p_k may only be feasible with a backhand orientation. Also, if the current paddle state p_s at the beginning of the strike is a backhand pose, the target p_k may only be reachable in time with a backhand target; switching from backhand to forehand may require more time. For these reasons, the land-ball policy considers both forehand and backhand executions for each p_k and picks the one whose $\hat{p}_k$ is closer to p_k.

8.1.3   Improved Policy with Cross-Entropy Method (CEM)

The landing target resulting from a given pre-contact paddle-motion and ball-motion states has a unimodal distribution, i.e. there is only one possible expected outcome. The inverse landing problem, however, may have multiple solutions. The inverse landing-prediction model captures the mean of all paddle-motion states that can send the ball to the target g. However, the mean action may be skewed and it may not do well.

A variant of the model-based land-ball policy uses an iterative refinement process based on CEM [24] to improve the final solution p_k returned from Algorithm 1. The refinement process performs a search in the space of paddle-motion states around p_k. Fig. 8.3 demonstrates the different stages of the land-ball policy with a forward CEM search. A population of solutions is created with the mean of p_k and an initial standard

deviation for each dimension in p_k. During each iteration of the algorithm, each individual in the population is evaluated using the forward landing-prediction model and its predicted target error is recorded. The predicted errors are used by the CEM algorithm to compute an updated mean and standard deviation for the next iteration. The process continues until either the maximum number of iterations is reached, or the predicted error falls below a fixed threshold. At that point the final mean value is returned as the target paddle-motion specifying the strike.

8.1.4   Evaluation

The land-ball policy is evaluated by launching random balls from the ball launcher and requesting the policy to land the ball at random targets. To allow for a fair comparison of the alternative striking policies discussed later in this chapter, in these experiments the forward and inverse landing-prediction models were trained only with position targets (without velocity). The dynamics models were trained both from a small dataset containing 7 000 trajectories, and a large dataset containing 140 000 trajectories to evaluate the impact of the number of samples on the accuracy of policies.

The robot always starts at a fixed stationary forehand pose with zero joint velocities. The ball is launched towards the robot from a random position and with random velocity. If the launched ball goes out or hits the net, the launch is repeated. In the simulation environment, the table is at the center of the coordinate system with its center point at $(0, 0, 0.76)$ m. The x coordinates increase away from the robot and towards the opponent, the y coordinates increase to the left side of the net, and the z-axis points up. The robot's base is at $(-1.8, 0, 0.76)$ m, i. e. 1.8 m away from the net. The initial ball position is chosen uniformly at random from a cube defined by $l_{\text{low}}(b_s) = (1.4, -0.3, 0.86)$ m, $l_{\text{high}}(b_s) = (2.0, 0.3, 1.26)$ m. The initial ball velocity is chosen uniformly at random between $v_{\text{low}}(b_s) = (-6, -0.5, 1.5)$ m/s and $l_{\text{high}}(b_s) = (-5, 0.5, 2.5)$ m/s. The landing target for the ball is chosen uniformly at random from a box 0.4 m away from the net and 0.1 m away from the table edges.

Table 8.1 shows the evaluation results for the two model-based land-ball policy variants with and without CEM search. A successful ball return requires that the robot strikes the ball over the net and onto the opponent's side of the table. The mean target errors are computed only over the balls that are successfully returned.

Visualizing the policy reveals that a high percentage of the failure cases happen
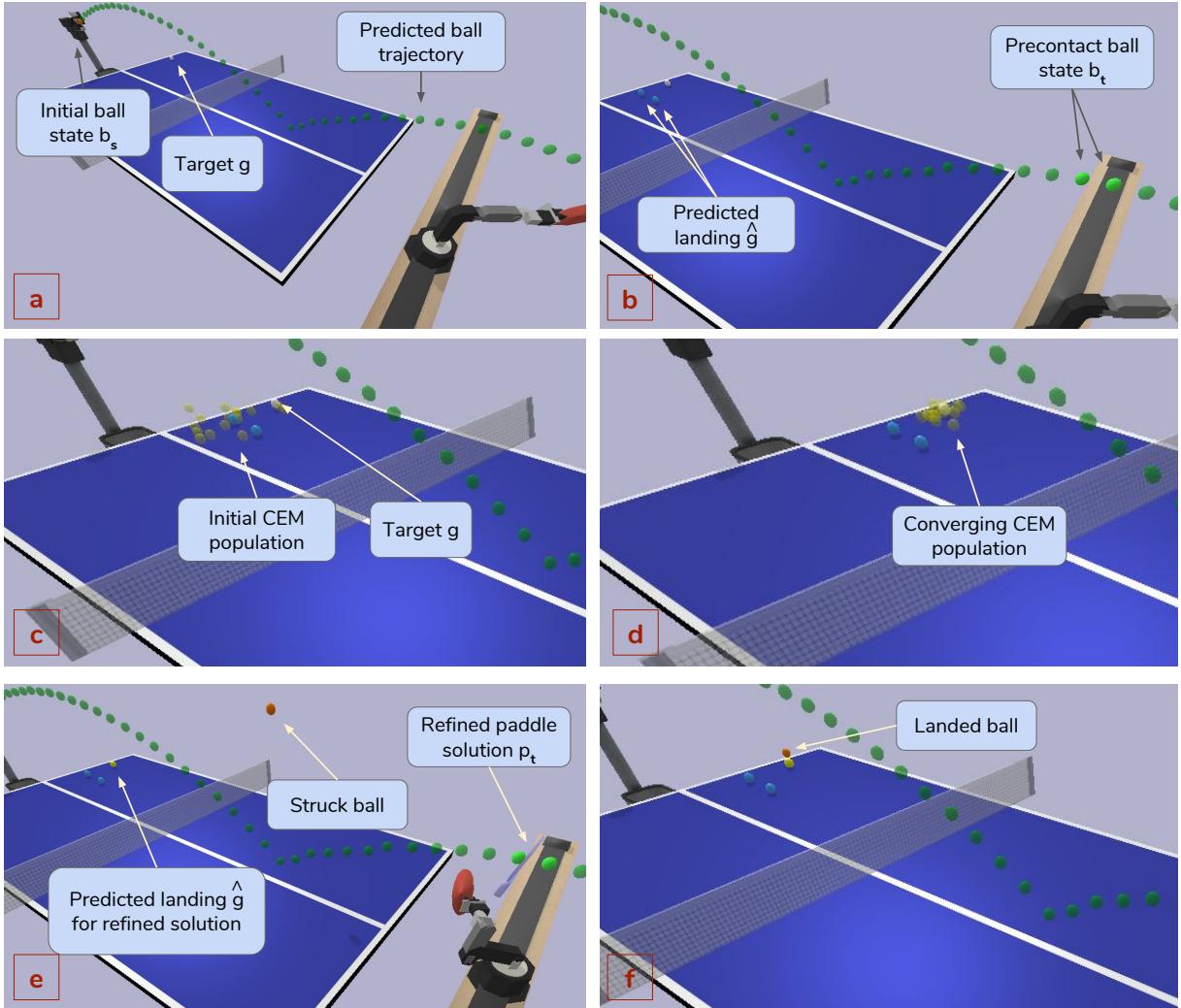
Figure 8.3: **Demonstration of the Land-Ball Policy with CEM Search. a)** See Fig. 8.2.a. **b)** See Fig. 8.2.b. **c)** Given the best candidate paddle-motion target p_t computed by the inverse landing model, a population of paddle-motion targets is created randomly around p_t. The translucent yellow balls visualize the predicted landing targets for each paddle-motion target in the CEM population. **d)** During each iteration of CEM, the population may get closer to the target g. **e)** Once the CEM population has converged, the final mean of the population of paddle-motion targets is passed as an action to the paddle-control skill. **f)** The ball lands on the table.

Method	Search	Data	Samples	Return Rate	Target Err
Model-Based	-	VR	7 000	88.0%	0.216 m
Model-Based	CEM	VR	7 000	86.7%	0.191 m
Model-Based	-	VR	140 000	89.6%	0.182 m
Model-Based	CEM	VR	140 000	90.8%	0.119 m

Table 8.1: **Evaluation of Model-Based Land-Ball Policies Trained with Human Demonstrations Collected in the VR Environment.** Mean ball return rate and mean landing target error computed over 1200 attempts with random targets. The policy uses dynamics models trained from data collected in the VR environment. The two variants of the policy (with and without CEM search) have similar return rates. When trained on the large dataset, the CEM variant has a higher target accuracy with an average error of about 12 cm, while the variant without CEM has an average error above 18 cm. The policy is sample-efficient, since even when trained on the small dataset it achieves an average target position error of about 20 cm.

because the ball hits the net. The data normalization process described in Sec. 6.3 normalizes all landing trajectories such that the x, y coordinates of the pre-contact ball state are set to zero. In doing so, the normalization process hides the position of the net from the model. The forward and inverse landing-prediction models operate as if the net did not exist. It is possible to add extra inputs to the landing models to specify a distance to the net. However, doing so would increase the number of dimensions, which might cancel some of the benefits of the data normalization process. This approach was not tested in experiments. In the hierarchical policy design, it is expected that the high-level strategy skill will account for the inefficiencies in the striking skills and will pick high-level actions that work around them.

Another failure happens when the launched ball goes straight to the table under the robot. An example of this failure case is shown in Fig. 8.4. In these cases, the robot is not able to execute its desired paddle strike because the paddle would collide with the table.

The land-ball skill provides a high-level abstraction over the agent's game-play during a single exchange with the opponent. This high-level abstraction does not reduce generality of behavior. Barring deceptive movements to hide the agent's intention from the opponent, any sequence of paddle actions can be reduced to the resulting landing state for the ball. In other words, the land-ball skill makes it possible to specify the agent's
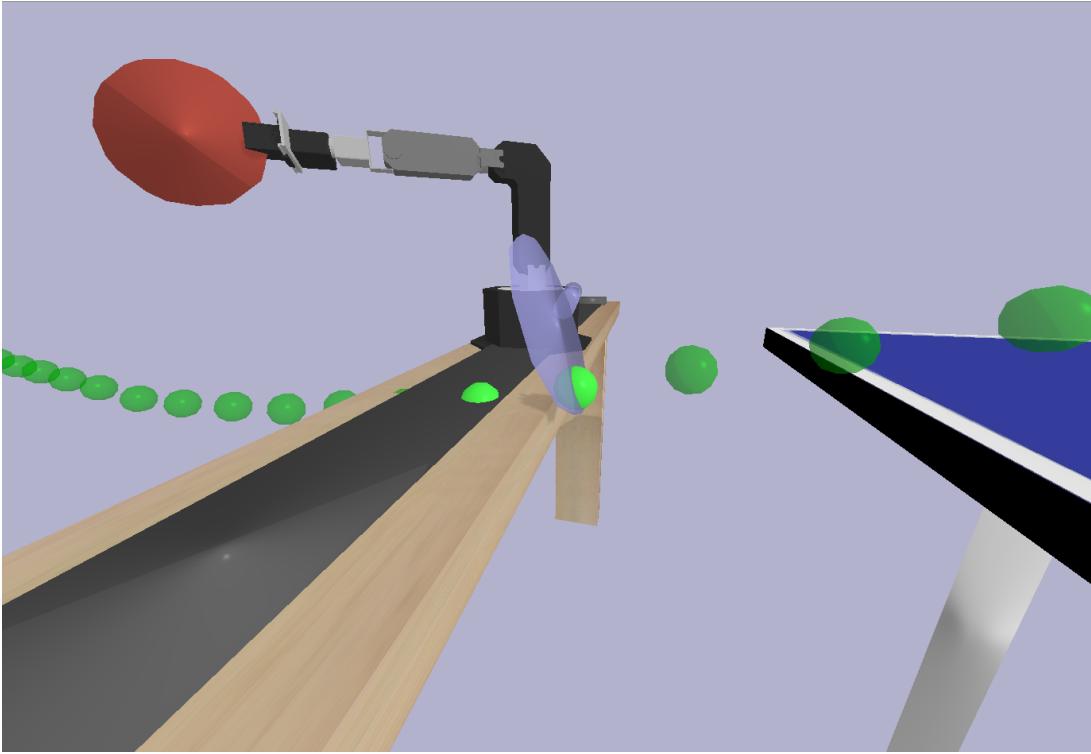
Figure 8.4: **Example Failure Case for Model-Based Land-Ball Policy.** At times, the trajectory of the launcher ball collides with the table under the robot assembly. The translucent paddle visualizes the requested paddle-motion target. Such balls are more difficult to return, as the planned paddle trajectory may collide with the table.

behavior by its desired outcome. Learning to use the land-ball skill is easier for the agents as its action space has fewer dimensions than a fully-specified target paddle-motion, yet its action space can specify complex behaviors.

8.2   Model-Based Land-Ball Trained with Robot Data

The analytic paddle controller executes strikes by executing paddle-motion targets at the moment of contact with the ball. These paddle-motion targets are learned from humans who can freely move the paddle around using the high degrees of freedom in their shoulder and wrist joints. In contrast, robots have few degrees of freedom. In particular,

the main robot used in these evaluations has only six. This observation raises the question whether requiring the robot to imitate human strikes imposes a restriction on the policy, as either the anatomy of the robot or the analytic paddle controller may not permit replicating the paddle motions demonstrated by humans. To answer this question, the model-based land-ball policies were also trained from strikes generated directly on the robot itself.

8.2.1 Data Generation

Generating successful strikes with the robot requires a functioning striking policy. However, such a policy does not exist at the beginning and it is not straight-forward to implement a hand-coded policy that can land the ball well, and at the same time demonstrate versatile strikes. As discussed in Sec. 7.4, generating random yet successful strikes is a hard problem. In this experiment, random strikes are generated by sampling fixed target velocities for each of six joints on the robot and passing them to the PID controller. The robot is always initialized to a fixed forehand pose at the beginning of the strike. The velocity targets are kept fixed during each strike. The range of velocity values is determined using a trial-and-error process to increase the likelihood of useful strikes.

A successful strike requires that striking paddle makes contact with the ball. Since the paddle strike is random, a cooperative ball launcher is used to launch the ball with the right velocity to meet a swinging paddle at a desired time. The position of the ball launcher is fixed, but the velocity of the launched balls is variable. Such a ball launcher has a real-world counterpart as well. Most commercial ball launchers have a user interface allowing the player to specify the desired velocity and spin of the ball. Implementing a cooperative launcher requires interfacing with the device and programmatically setting the desired launch attributes for the ball. To achieve contact between the ball and the paddle, this experiment trains and uses two additional dynamics models:

1. **Forward low-level paddle-control model**: Similar to what is described in Sec. 7.4, a model is learned to predict the trajectory of the paddle-motion states resulting from executing a set of fixed joint velocities. The model is implemented as a recurrent neural network, which given the velocity targets, predicts the full paddle-motion state over subsequent timesteps assuming that the robot always starts from a fixed pose.

91

2. **Inverse ball-launcher model**: The cooperative launcher uses the inverse ball-launcher model. The inputs to this model are a desired ball position observation in the future and a corresponding time. The outputs are the 3D launch velocity vector, such that launching the ball from the fixed launcher position with that velocity will result in a ball that bounces on the table and reaches the desired position at the desired time. The inverse ball-launcher model is trained from trajectories of randomly-launched ball.

8.2.2 Evaluation

Once the two above dynamics models are trained, they are used to generate random strikes and collect landing trajectories. For each sample, a set of fixed velocities are sampled for the robot joints. The forward low-level paddle-control model is used to predict the trajectory of the paddle. A random paddle position that lies within the same 20 cm band described in Sec. 8.1 is picked from the trajectory. Then, the inverse ball-launcher model is used to compute a velocity for the launcher ball such that the launched ball reaches that paddle position at the same time as the paddle. This process is likely to create contact between the ball and the paddle. Moreover, the range of random joint velocity targets are hand-tuned in a way that some of the returned balls reach the opponent's side of the table.

Fig. 8.5 shows the distribution of landing positions resulting from a sample set of landing trajectories generated with the above setup. Since many of the returned balls do not land successfully, a total of 900 000 landing trajectories are generated to have plenty of landed balls to learn from. The landing trajectories collected in this process are used to train the forward and inverse landing-prediction models described in Chapter 6. A land-ball policy using these dynamics models is evaluated using the same evaluation setup outlined in Sec. 8.1.4. Table 8.2 shows the evaluation results.

Comparing the mean target errors with the policy from Sec. 8.1 shows that this policy achieves a lower accuracy for the land-ball task. The training data for this policy uses only strikes that are guaranteed to be feasible on the robot since they were generated by the robot in the first place. However, the higher accuracy of the policies trained with the VR data suggest that the analytic controller is able to execute arbitrary strikes demonstrated by the humans as well. The most likely reason behind the lower accuracy in this policy is that the random strikes generated using fixed joint velocity targets are not general enough to capture all possible useful motions.
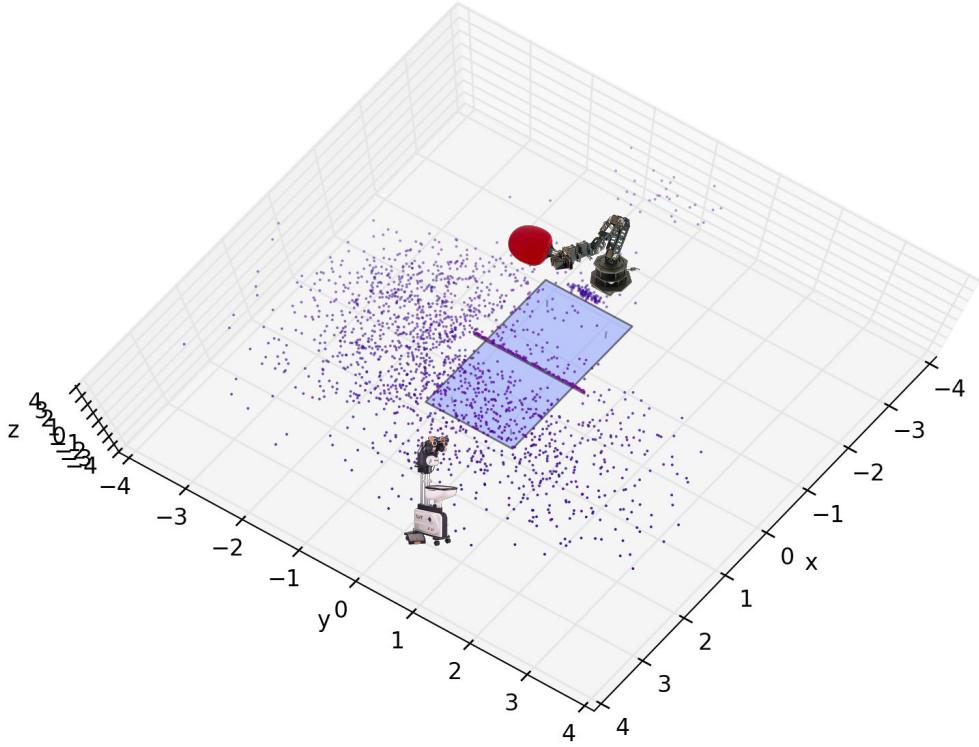
Figure 8.5: **Distribution of Landing Positions in Landing Trajectories Generated on the Robot.** The blue dots show the eventual landing location of the trajectories in the training dataset. For balls that do not hit the table, the dot shows the point where the ball trajectory intersects with the surface plane of the table. The visualized distribution shows that a good percentage of the returned balls do land on the opponent's side of the table. The visualization also shows that hitting target right behind the net is difficult for the robot, which is why the random landing targets are chosen to be at least 40 cm behind the net. The landing trajectories which are generated by the robot are used to train alternative dynamics models. These models are used to evaluate whether training the models only on strikes that are known to be feasible on the robot improve performance. Experiments show that this is not the case, thereby showing that requiring the analytic controller to reproduce striking motions demonstrated by humans does not limit the striking skill's performance.

Method	Search	Data	Return Rate	Target Error
Model-Based	-	Robot	92%	0.191 m
Model-Based	CEM	Robot	94%	0.203 m

Table 8.2: **Evaluation of Model-Based Land-Ball Policies Trained with Data Generated on the Robot.** Mean ball return rate and mean landing target error computed over 600 attempts with random targets. The policy uses dynamics models trained from 900 000 landing trajectories collected on the robot itself. Comparing the results with Table 8.1 shows that these policies have lower target accuracies than policies trained from the human demonstrations collected in the VR environment.

8.3 Model-Free Land-Ball Policy

The model-based land-ball policies employ the dynamics models that are trained either from human demonstrations or random strikes generated on the robot. In order to assess the impact of dynamics models on sample-efficiency and accuracy of the land-ball policies, this section describes and evaluates an alternative implementation of the policy which uses model-free reinforcement learning to train the policy from scratch. The experiments help put the sample-efficiency and accuracy of the model-based land-ball policies into perspective.

8.3.1 Training

Unlike the hierarchical controller used in model-based land-ball policies, the model-free land-ball policies are expected to control the robot at the joint level and choose actions at every timestep. Since the land-ball task is more difficult to learn under these conditions, the model-free agents are also trained and evaluated on a simpler task whose goal is to just successfully return the ball to the opponent's side. The reward functions used for training the model-free policies are:

1. **Return-Ball**: The agent is expected to return the ball over the net successfully so that it bounces on the opponent side of the table. This task has a binary binary reward δ_{return} indicating success or failure.

2. **Land-Ball**: The agent is expected to land the ball at a randomly-picked target position (with no specified target velocity). The reward for this task is specified

by $\delta_{\text{return}}(3 - \|l(b_g) - l(g)\|_2)$, where $l(b_g)$ denotes the observed landing position of the ball at the moment of contact with the table and $l(g)$ denotes the target landing position. So, if agent the cannot return the ball, it receives a reward of zero. Otherwise, it receives a positive reward depending on how close it gets to the target.

8.3.2 Evaluation

This section describes the evaluation results obtained from learning model-free policies using Proximal Policy Optimization (PPO) [25] and Augmented Random Search (ARS) [14] algorithms.

The model-free policies use feed-forward neural networks with a single hidden layer of 20 units. The input observation vector includes the position and velocity of the ball and robot joints. When trying to send the ball to a particular target location, the observation also includes the coordinates of the target position. The output of the policy is the six-dimensional vector containing target robot joint velocities.

Learning the task with PPO was not very successful. When trained on the simpler return-ball task, the PPO agent could achieve a successful ball return rate of 40%. Training the PPO agent on the more difficult land-ball task did not produce effective policies.

The ARS agent uses Algorithm 2 by Mania et al. [14], with normalized observations, step-size 0.05 scaled by reward standard deviation, perturbation standard deviation 0.05, and updates limited to the five top-performing perturbations out of the 30 evaluated. The ARS algorithm applies perturbations to the policy weights and evaluates each perturbation using a rollout. For the return-ball and land-ball tasks, each rollout corresponds to evaluating the policy on a new random target and a new random launcher ball in the environment. Since the parameters of the launcher ball and landing target are different from episode to episode, the tasks are highly stochastic. So, the algorithm is modified to evaluate each perturbation using up to 15 rollouts rather than a single rollout. This increase in the number of evaluations led to faster training progress and better performance, possibly due to choosing policy update directions that perform well over the randomness in the environment.

Fig. 8.6 shows the training rewards received by the ARS agent on the return-ball task under different number of evaluations k per policy perturbation. The figure shows

four runs of the algorithm for each value of k. Even though higher values for k increase the number of evaluations needed, they improve the learning performance. When each policy perturbation is evaluated multiple times, the stochasticity in the environment does not influence the direction of policy weight updates as much.
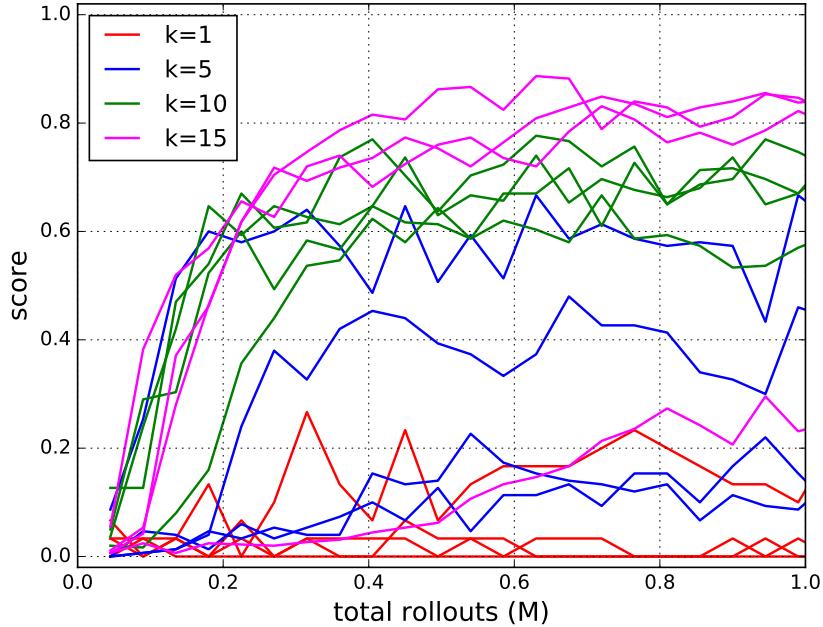

Figure 8.6: **Training Model-Free Return-Ball Policy with ARS.** The figure shows training score vs. total number of rollouts on the binary-reward task of successfully returning the ball, for ARS runs with a different number of evaluations k per perturbation (indicated by colors). Evaluating each perturbation using multiple rollouts improves the training score, as the algorithm chooses update directions that perform well over the randomness in the environment.

Fig. 8.7 shows the evaluation rewards received by four different runs of the ARS training with $k = 15$. ARS stopped making progress after ~2M rollouts, and the best policy (out of four runs) succeeded 88.6% of the time. This experiment shows that ARS is able to discover a better model-free return-ball policy compared to PPO.

Fig. 8.8 shows the training rewards received by the ARS agent on the land-ball task. The agent converges after ~3M rollouts. Evaluating the best policy at the end of training shows a ball return rate of 88% and an average landing distance of 0.4 m from the
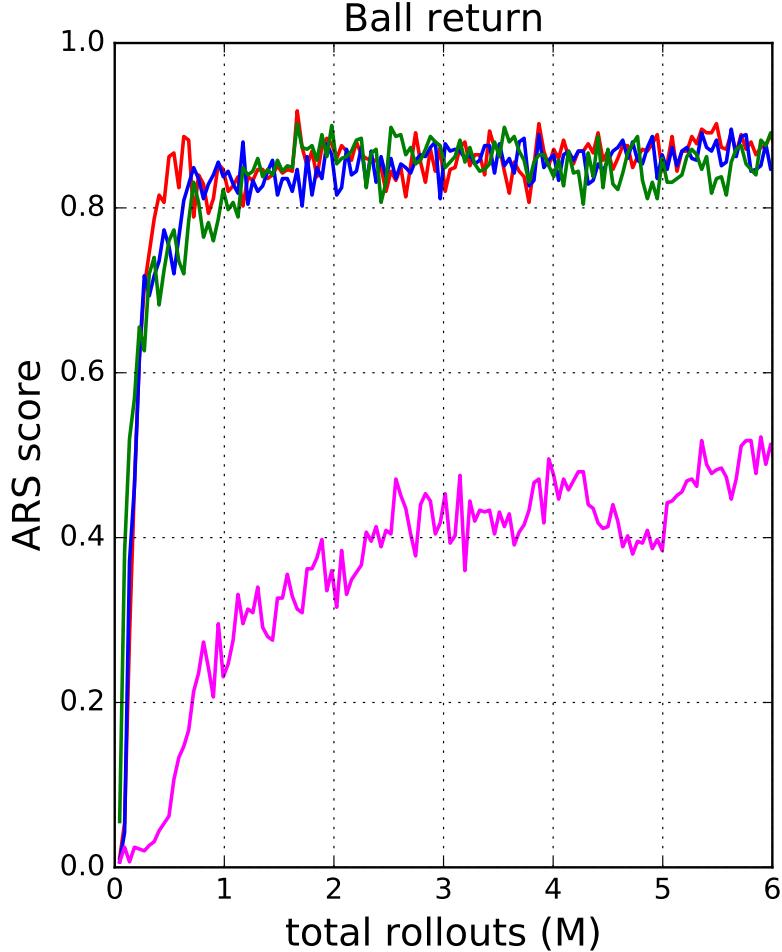
Figure 8.7: **Evaluation of Model-Free Return-Ball Policy Trained with ARS.**
The four lines show the evaluation scores received from four different runs of ARS on the
return-ball task. All runs use $k = 15$. Learning the policy with model-free RL requires ∼1-
2M episodes. Such a high number of samples prevents using such algorithms on physical
robots in the real world.

target. While this policy is fairly inaccurate, it is better than random. In comparison, the
best policy trained on the return-ball task achieves a distance of $0.52\,\mathrm{m}$ from the target
on average.

Table 8.3 summarizes the evaluation scores achieved by the model-free agents.
Overall, ARS produced substantially better results than PPO; policies trained using ARS
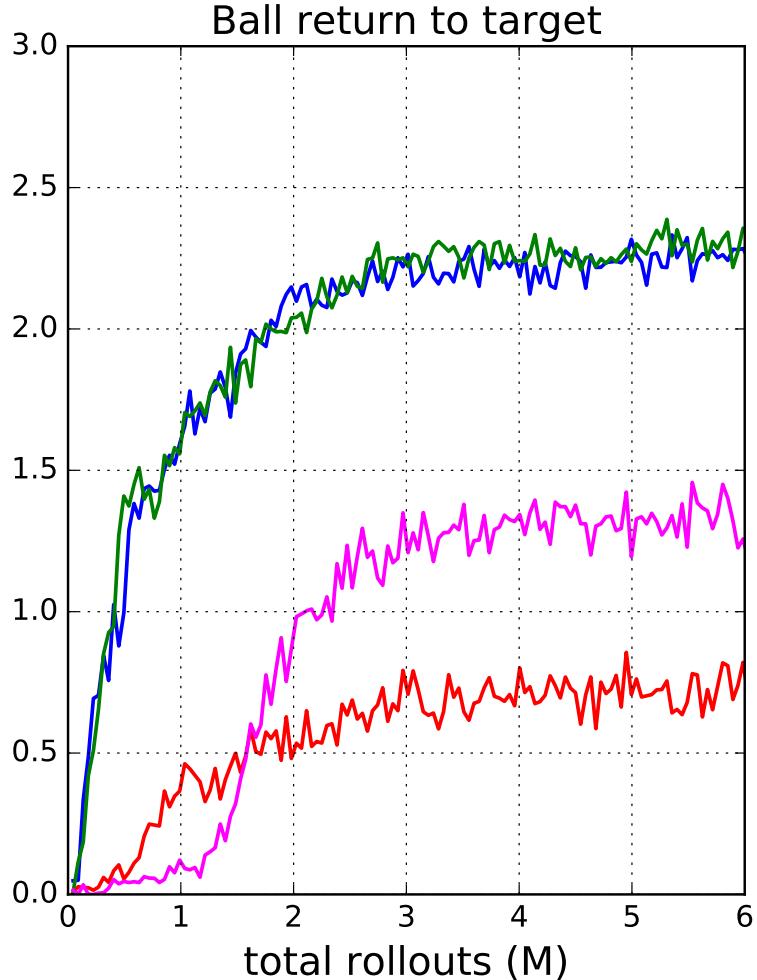
Figure 8.8: **Training Model-Free Land-Ball Policy Trained with ARS.** The four lines show the ARS training score vs. total number of rollouts on the task of returning the ball to a particular target location. All runs use $k = 15$. Two out of the four runs perform better and achieve a mean reward of between 2.0 and 2.5, which as the evaluation show, corresponds to a landing accuracy of about 40 cm. Moreover, the model-free RL algorithm requires $\sim$3M episodes, which makes it impractical for the real world.

were able to return the ball successfully $\sim$88% of the time, while those trained using PPO only succeeded $\sim$40% of the time. However, the ARS agent used $\sim$3M training episodes to learn the land-ball task. It is not feasible to run such a large number of

training episodes on physical robots in the real world. Comparing the performance of the model-free policies to the model-based hierarchical policies in Table 8.1 shows that the model-based approach produces policies with much lower error (12 cm vs. 40 cm) and using far fewer samples (140 000 vs 3M). Moreover, the samples used for model-based policies come from human demonstrations, which are far easier to obtain than samples collected with the robot itself.

Method	Algorithm	Samples	Return Rate	Target Error
Model-Free	PPO	0.1 M	40%	-
Model-Free	ARS	3 M	88.6%	0.4 m

Table 8.3: **Evaluation of Model-Free Land-Ball Policies.** Mean ball return rate and mean landing target errors for random targets. Comparing the results with Table 8.1 shows that this policy has a lower target accuracy.

8.4   Model-Based Hit-Ball Policy

The model-based land-ball policies require three trained dynamics models to operate. Using the dynamics models make the policy implementations sample-efficient. On the other hand, if the models are not updated with additional trajectories collected using other policies, the policies that depend on those models would stay limited in their behavior.

8.4.1   Policy Implementation

The hit-ball policy is an alternative striking policy that can execute more versatile striking motions. It uses only one dynamics model (the ball-trajectory prediction model). Using the ball-trajectory model makes it easy for the policy to ensure contact with the ball. The position of the paddle is simply set to the prediction position of the ball at the desired time of contact. The other parameters of the paddle-motion target (paddle orientation, linear and angular velocity) are given as inputs to the hit-ball policy. So, a higher-level skill like the strategy skill can use the hit-ball policy to execute arbitrary strikes without consulting any additional models to decide a target paddle-motion state.

As described in Sec. 4.3.2, the hit-ball task is specified as:

$$\pi_h(l_x(p_t), N(p_t), v(p_t), \omega(p_t) \mid b_s) = \pi_p(t, p_t \mid p_s). \tag{8.1}$$

where $l_x(p_t)$ denotes the desired contact plane. The policy is instructed to hit the incoming ball when it reaches the plane $x = l_x(p_t)$.

Fig. 8.9 illustrates the implementation of the hit-ball policy. The ball-trajectory prediction model is used to predict the incoming ball's trajectory based on the ball's initial motion state b_s. The policy then intersect the ball trajectory with the the desired contact plane $x = l_x(p_t)$ and picks the point b_t that in the ball trajectory that is closest to this plane. This point determines the contact time t and the full position of the paddle $l(p_t)$ at the time of contact. The other attributes of the paddle's motion state $N(p_t), v(p_t), \omega(p_t)$ are given as inputs to the hit-ball policy. Together with the computed paddle position, they fully specify the paddle-motion target p_t, which is passed as a high-level action to the paddle-control skill.

8.4.2   Evaluation

The hit-ball policy can not be easily evaluated in isolation since it depends on a higher-level policy to specify the orientation and velocity of the paddle at the time of contact. This policy is used in the next chapter by a strategy agent to play whole table-tennis games. The hit-ball policy allows the strategy agent to discover strikes that are particularly useful for cooperative or adversarial games, beyond what the model-based land-ball policy produces.

8.5   Conclusion

This chapter described multiple implementations for the land-ball policy. Experiments showed that:

- The model-based land-ball policy trained from VR data is very sample-efficient and can hit ball targets with a mean position error of only $12\,\mathrm{cm}$.

- Training the model-based land-ball policy with data collected directly on the robot does not improve its performance. So the analytic paddle controller is able to execute arbitrary strikes demonstrated by humans.
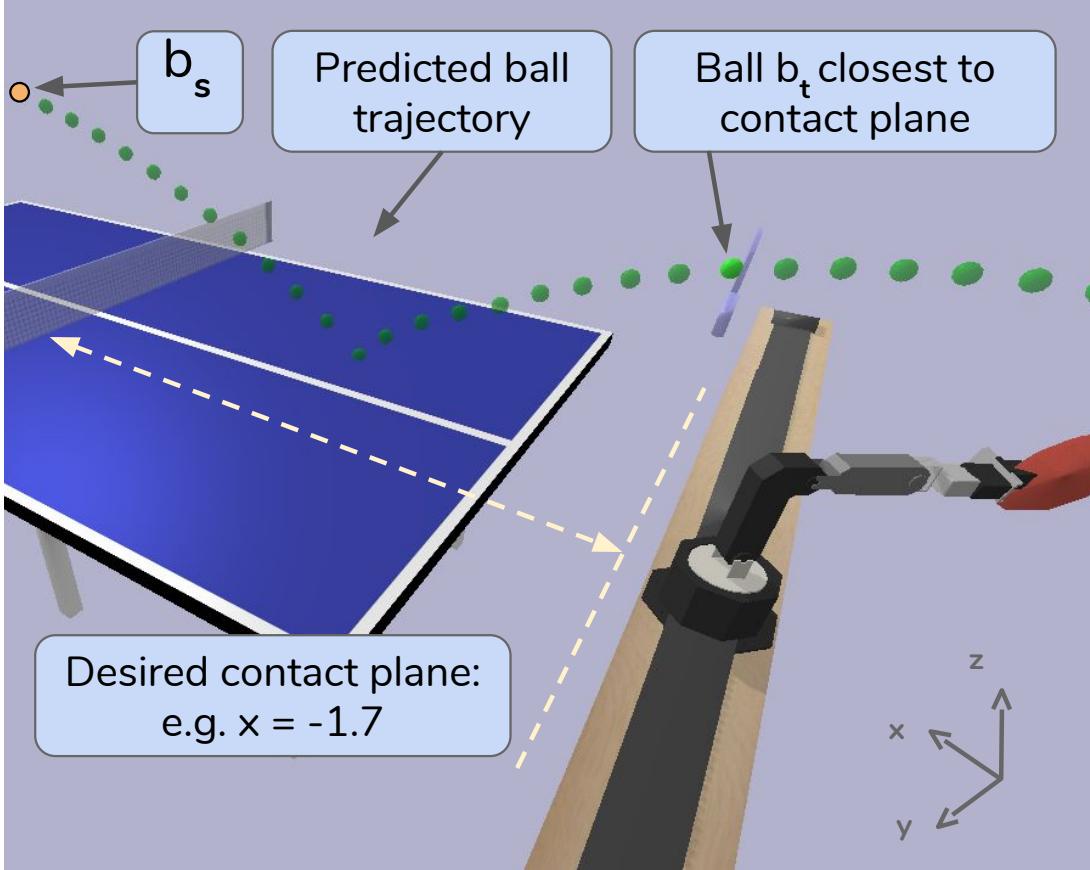
Figure 8.9: **Model-Based Hit-Ball Policy.** The action space of the hit-ball skill includes a desired contact plane, which indirectly specifies a target point for striking the ball. Given the current ball-motion state b_s, the hit-ball policy predicts the future ball trajectory. The point b_t in the predicted trajectory that is closest to the requested contact plane is picked as the striking target. This point, which is visualized as the light-green ball, specifies both the target position for the paddle $l(p_t)$, and the target time t. The other components of the paddle-motion target, including its normal vector, and linear and angular velocities are given as inputs to the hit-ball policy. Using the simple ball-trajectory model helps the hit-ball policy make contact with the ball easily. On the other hand, accepting any orientation or velocity for the paddle allows the hit-ball skill to execute arbitrary paddle strikes. Thereby, the hit-ball skill is more flexible than the land-ball skill.

- It is difficult to learn the land-ball skill with model-free learning methods that

control the robot at joint level. Such model-free policies require about 3M training samples and achieve a target landing accuracy of about 40 cm.

Although the land-ball skill itself can capture any game-play behavior, the canonical land-ball policy implemented in this dissertation uses dynamics models trained only from human demonstrations. So, the implemented land-ball policy only executes paddle strikes that were used by humans. The trained models might generalize to use blended strikes within the space of demonstrations, but the models cannot output strikes that lie completely outside the space of demonstrations. For example, if the humans have never hit the ball from below so it goes high up, the trained models cannot produce or evaluate such strikes. On the other hand, this choice of implementation is not a limitation on the approach. It is possible to fine-tune the landing models trained from human demonstrations with data collected from other striking policies, as done in the dagger method [23]. Such an approach can combine the benefit This approach is left as part of future work.

The next chapter describes how the striking policies can be used by the top-level strategy skill in the task hierarchy to play cooperative and adversarial table-tennis games.

Chapter 9

Learning Strategy with Self-Play

This section describes how the strategy skill is trained. It then evaluates different variants of the strategy skill that use different striking skills.

9.1 Approach

The striking skills discussed in chapter Chapter 8 make it possible to execute high-level paddle actions that either send the ball to a desired target location (land-ball skill), or make contact with the ball with a desired paddle orientation and velocity (hit-ball skill). Similarly, the positioning skill described in Sec. 7.5 can be used to move the robot from any motion state to a desired pose in minimum time so that they robot can be ready for responding to the next shot from the opponent. However, the striking and positioning skills are parameterized skills. They should be given goals to execute. Playing successful table-tennis games using the striking and positioning skills requires deciding appropriate targets for them during each exchange of the ball with the opponent. It is possible to operate the underlying skills using hand-coded targets. However, such a hand-coded implementation would be very limiting in exploiting the diverse behaviors that are possible in the striking and positioning policies. So, this method uses a model-free RL algorithm to learn diverse game-play strategies.

9.1.1 Reinforcement Learning

The task hierarchy discussed in Chapter 4 addresses this problem by including a top-level strategy skill whose only job is to pick targets for the striking and positioning skills. An effective strategy policy requires complex reasoning about the opponents, which is difficult to do in a model-based way. So, the strategy skill is learned with model-free reinforcement learning. The decomposition of the tasks and the abstractions offered in

the skill hierarchy makes it possible to combine this model-free strategy layer with model-based striking skills and the analytic robot controller in a single agent.

In the method implemented in this disseration, the strategy skill is the only skill that needs to be trained on the robot. Although model-free reinforcement learning is not sample-efficient, the design of the hierarchical policy simplifies learning the strategy layer, thereby making model-free learning applicable to the problem. Since the strategy skill does not need to be aware of the internal implementation of the underlying skills, its inputs and outputs can stay low-dimensional. Also, since the striking and positioning skills have long time horizons spanning over an entire ball exchange with the opponent, the strategy skill gets to observe the game at a high-level. Therefore, it can focus on the high-level game-play. If the strategy skill picks a bad landing target or a bad waiting position in a game, it receives the negative reward immediately in the next step. The long time horizon helps eliminate the reward delay, which can make learning this skill more efficient. Since the skill can be trained after the striking and position policies are developed, all the training episodes used when learning the strategy can focus on exploring the space of strategies, rather than also being concerned with learning striking and positioning policies.

9.1.2 Cooperative and Adversarial Rewards

This dissertation considers two types of table-tennis games: cooperative and adversarial. In a cooperative game, the objective of the two players is to keep the game going as long as possible. In this mode, the players receive one point each time they land the ball anywhere on the other player's side. The cooperative reward function also gives the players one point each time they hit the ball with the paddle. Since the land-ball and hit-ball policies automatically make contact with the ball, they do not benefit that much from this reward shaping mechanism. However, this chapter also includes experiments with learning a strategy directly over the paddle-control skill. Reward shaping is useful for learning such a low-level strategy. So, it is included for all agents to make comparison of rewards easier. With reward shaping, a cooperative agent can receive a maximum reward of +4 per step.

In adversarial games, the objective is to win the rally. Each player tries to land the ball over the opponent's side in a way that the opponent cannot successfully return the ball. In this mode, the player who wins the rally receives one point. Players also receive a reward of 0.1 for making contact with the ball or successfully landing the ball anywhere on the opponent's side.

9.1.3 Self-Play

Since the strategy skill does not have a specific target, it is best to learn the skill using self-play. The only objective for the strategy skill is to maximize the reward. Since the reward received depends also on the behavior of the opponent, the skill needs to adapt to the behavior of the opponent. This goal is achieved in a self-play setup, which is shown in Fig. 9.1. Two identical robot assemblies are mounted on the two sides of the table. Since the agent policies do not support serving the ball, the rally is always started with a ball launcher. There is a launcher behind each robot and the rally starts randomly from one of the two ball launchers. When the ball goes out of play, the next ball is launched and a new rally starts.
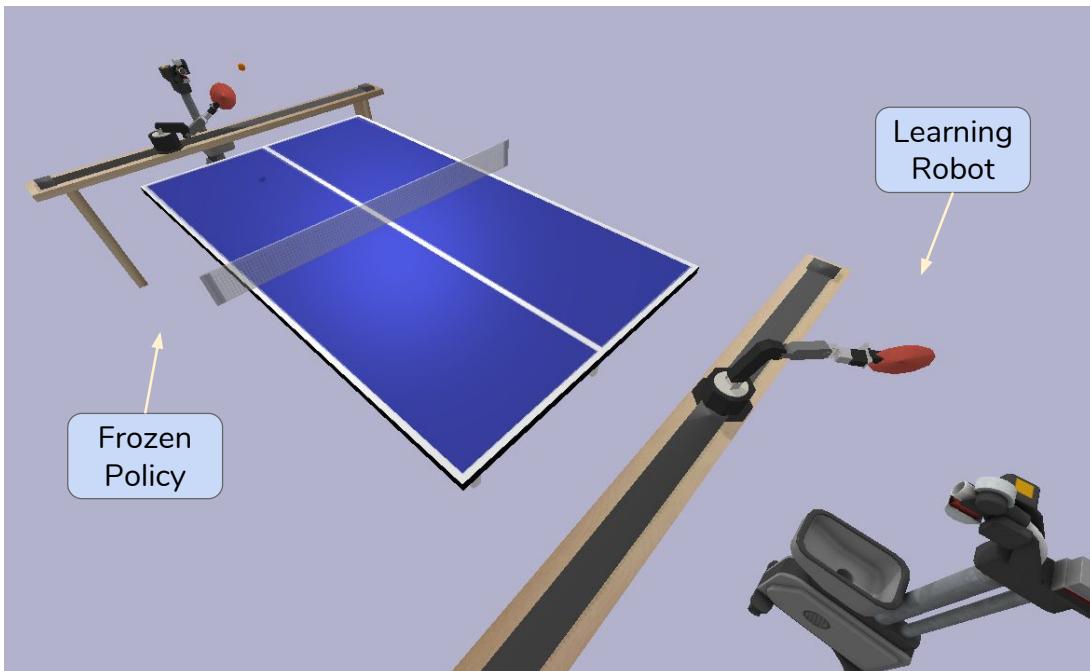
Figure 9.1: **Self-Play Setup.** Two robots play cooperative or adversarial games with each other. The rallies are started randomly by one of the two ball launchers behind the robots. Only one of the two robots is learning the strategy policy. During each self-play level the learning robot plays against a its own frozen policy from the end of the previous level. As more self-play games are played, the strategy policy learns to adapt and respond to its own behavior.

Of the two robots playing against each other, only one is actively training. The

other robot is playing using a frozen policy. Self-play learning is done over a number of levels. During the first level, the learning agent plays against an initial fixed policy. After the end of the first level, the learner's policy is frozen and transferred over to the opponent. So, during level i, the learner plays against its own frozen policy at the end of level $i - 1$.

9.1.4 Observations and Actions

Each step in the environment lasts over an entire ball exchange. Each ball exchange can include two strikes, one by the learner and another by the opponent. In the current implementation, the strategy skill receives only the current ball-motion state b_s as the observation. Preliminary experiments showed that adding other observations like the agent's paddle position $l(p_t)$ or the opponent's paddle position do not improve performance in the current implementation. It is likely that in a setup where the agent can make more than one decision per exchange including such additional observations would be useful. As actions, the strategy skill produces targets for the striking skill and the positioning skill.

In the current implementation, the strategy skill specifies a waiting position for the robot only once per exchange. This choice is made because the agent does not receive observations from the opponent's movements as the opponent approaches the ball to hit it back. Therefore, the strategy agent does not observe opponent's position when deciding on a target for the positioning skill. However, the behavior of the opponent and how it approaches to return the ball may contain clues for the other agent as to how to best position itself to be prepared for the return shot. In an alternative implementation where the positioning skill is invoked in a continuous manner, it would make sense to also include the opponent's position in the observations.

9.2 Training Setup

All strategy policies except for the joint-level strategy policy are implemented using the TensorFlow implementation of batched PPO [8]. The PPO agents use feed-forward policy and value networks. Both the policy and value networks have two hidden layers of size 10 each. The policy network produces a mean and a standard deviation value for each action dimension. Therefore, the learned policies are stochastic. The stochastic actions

are active for the frozen opponent policy and for evaluation of the learner's policy as well. In addition to the environment rewards, the algorithm also uses an entropy reward with a coefficient of 0.1 to encourage more randomness in actions. This stochasticity is useful for preventing the self-play agents from converging to very narrow policies. The learning rate for the policy network is 10^{-4} while the learning rate for the value network is 10^{-3}.

Each self-play level consists of 2400 training steps, which can span over any number of episodes. The method uses PPO with a batch size of 24, where each worker trains for a 100 steps. Each training and evaluation episode runs for a maximum of 10 steps. So, if a cooperative game reaches 10 steps, it is terminated even if the rally is still going on. Terminating the episodes early allow the agent to get more training time over difficult initial conditions where the first launcher ball is more difficult to return.

9.3 Land-Ball Strategy

The canonical model-based land-ball policy in this dissertation is trained solely from human demonstrations and does not use any training episodes on the robot. Given a landing target g, the land-ball policy consults the dynamics models to find a paddle strike that can land the ball at g. Even though the land-ball policy is limited to using human paddle strikes, it can still execute landing targets that were never tried in human demonstrations. The normalization process used in training the dynamics models makes them capable of executing any targets resulting from translation and rotation of observed landing trajectories. Training a strategy policy over the model-based land-ball skill amounts to learning a better approach to playing the game using the same ball-handling technique demonstrated by human players.

When playing adversarial games, a common strategy is to aim for targets near the right or left edges of the table. However, in the action space of the land-ball policy, these areas lie on different regions of the action space. So, even with the entropy terms, a strategy is likely to aim only for the right or the left side of the table. To address this problem, when deciding land-ball targets, the land-ball strategy produces a target position $l(g)$, a target speed $|v(g)|$, and a probability p_{flip} for flipping the target across the y dimension (left-right) over the table. The flip probability allows the strategy to specify targets near the right and left edges of the table without having to generate landing coordinates explicitly for both right and left sides.

Fig. 9.2 shows the training progress for the land-ball strategy for cooperative games. The plot spans over 35 self-play levels and shows the average length of a co-operative episode in evaluation. The cooperative agent converges to an episode length of eight after about 10 self-play levels (24000 training steps.) The policy is initialized to have a mean action corresponding to a target near the center of the opponent's side. As shown in the plot, the strategy skill can sustain a mean episode length of four right at the start of training. This high performance shows that the model-based land-ball policy is effective. Since the analytic paddle controller and the model-based land-ball policy do not use the robot for training, this experiment demonstrates zero-shot learning in cooperative table tennis. Fig. 9.3 shows the average evaluation reward for the cooperative land-ball strategy. The average reward equals the episode length multiplied by 4.0.

After about 10 self-play levels, the evaluation reward does not improve much more. The agent trained at the end of the 35th level is evaluated against itself in 240 cooperative episodes. In these episodes, the length cap is raised to 1000, so the agents are allowed to continue the rally for up to 1000 exchanges. Fig. 9.4 shows the histogram of episode lengths observed in these evaluation games. The mean episode length is 17 steps, with the maximum length reaching 124.

The land-ball striking skill can also be used to learn adversarial strategies. Fig. 9.5 shows the training progress for an adversarial land-ball strategy. After 77 levels, the mean episode length reaches about 2.5, meaning that on overage the episode ends after five strikes from the two robots. Visualizing the trained strategy shows that the agents try to hit the ball with high velocity or aim closer to the edges of the table. Fig. 9.6 visualizes an example.

Although the land-ball strategies can discover game-plays that are different from the game-plays employed by humans, the agents are limited to the striking techniques that are present in human demonstrations. The next section describes how a hit-ball strategy can learn new striking techniques as well.

9.4 Hit-Ball Strategy

The hit-ball skill helps the agent make contact with the ball by setting the target position of the paddle to be equal to the predicted position of the ball when it crosses a desired contact plane. The land-ball skill just expects its user to specify the paddle orientation (normal) and linear and angular velocities at the moment of contact. Learning
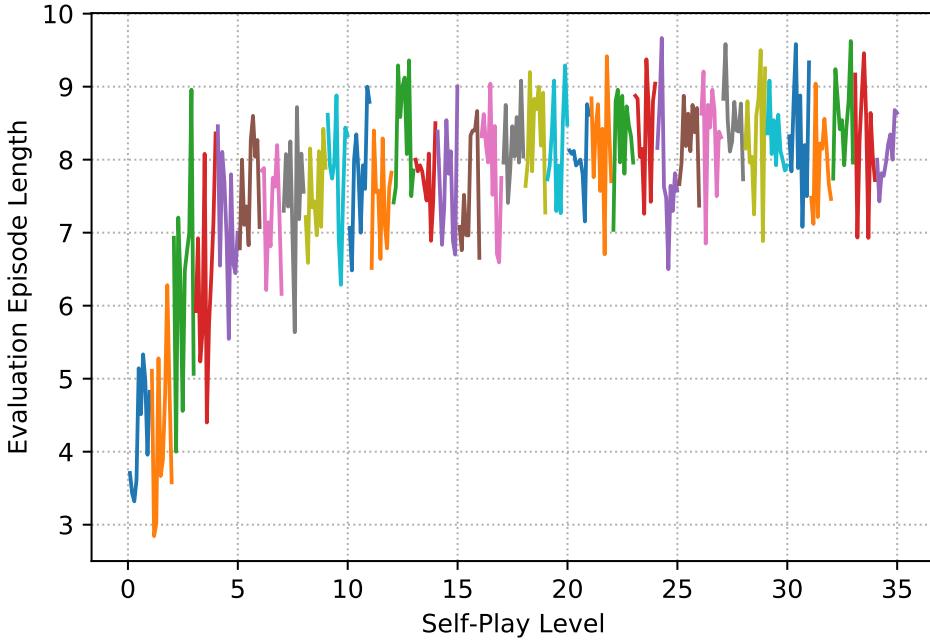
Figure 9.2: **Self-Play Learning of Cooperative Land-Ball Strategy.** Mean evaluation episode length over 35 self-play levels. Each colored line segment corresponds to a different level. The vertical axis shows the average length of a cooperative episode in evaluation. Each point is an average over 240 evaluation episodes. The maximum episode length is 10. The cooperative agent converges to an episode length of about eight after about 10 self-play levels (24 000 training strikes.)

a hit-ball strategy is more difficult, since hit-ball's action space has more dimensions than that of land-ball. On the other hand, a hit-ball strategy is free to use arbitrary paddle strikes.

Fig. 9.7 shows the training progress for a cooperative hit-ball strategy. The plot spans over 233 self-play levels. The hit-ball strategy starts with a mean episode length of one. After about 75 self-play levels, the mean episode length reaches four, which is the value achieved at the start for the land-ball strategy. After about 125 levels, the hit-ball strategy achieves a mean episode length of eight, and then it surpasses it to get close to nine on average. At times, the agent can achieve a the maximum episode length of 10
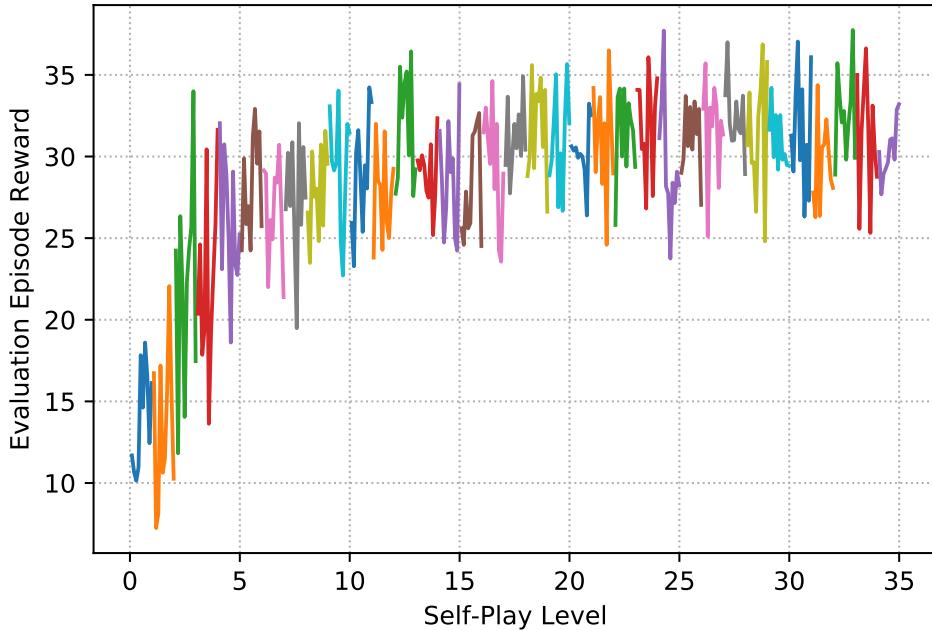
Figure 9.3: **Average Evaluation Rewards for Cooperative Land-Ball Strategy.** Mean evaluation reward over 35 self-play levels. The average reward equals the episode length multiplied by 4.0.

over 240 evaluation episodes.

Learning a cooperative hit-ball strategy is quite slower than learning a land-ball strategy, so it is less sample-efficient. On the other hand, it can achieve higher rewards. Fig. 9.8 shows the histogram of episode lengths observed in 240 evaluation games run with the final cooperative hit-ball policy. The agent achieves a mean episode length of 111.9 with the maximum reaching 600. Visualization of the policies leads to an interesting observation. The agents have learned to hit the ball rather slowly and with a paddle motion that causes it to land at about a 45 degree angle on the other player's side. Such high balls are less likely to hit the net or go out. Moreover, after bouncing on the table, the ball usually reaches the top of its arc in front of the other robot, making it easier for the other player to return it.

In contrast, Fig. 9.9 shows the training progress for an adversarial hit-ball strategy.
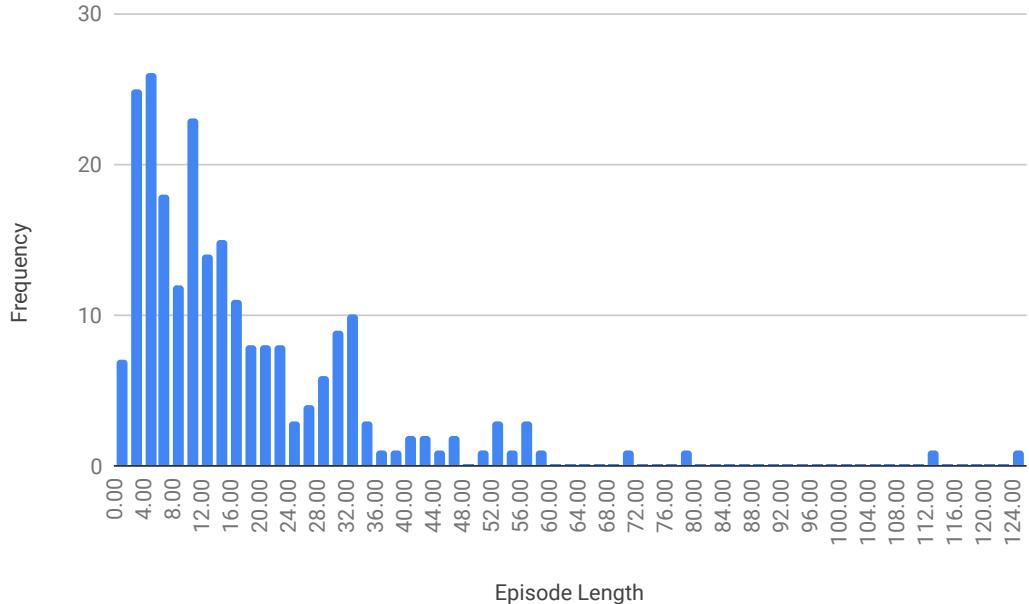
Figure 9.4: **Histogram of Episode Lengths for Cooperative Land-Ball Strategy.**
Distribution of episode length over 240 test episodes with an episode length cap of 1000.
The land-ball strategy achieves a mean episode length of 17.4 with a standard deviation
of 17.1.

Very quickly, the mean episode length drops to one, signifying that the adversarial games
do not last long. Fig. 9.9 shows the mean evaluation rewards received by the learning
agent during adversarial training. The mean episode reward reaches close to one at the
beginning, which means the learning agent can easily exploit the opponent with the frozen
policy. With more training, the reward becomes more balanced. Visualizing the trained
strategy shows that the agents have discovered a way to send the ball over the head of
the opponent. In most episodes, the agent who gets to act first, hits the ball with enough
vertical force that after bouncing on the table it cross above the reachable space of the
other robot. Fig. 9.11 visualizes and example of this behavior.

The experiments show that a hit-ball strategy can discover more effective coopera-
tive and adversarial strategies at the expense of lower sample-efficiency. The hit-ball skill
supports executing arbitrary strikes and is as general as a striking policy can be. It uses
only a ball-trajectory prediction model that is easy to train. The experiment in the next
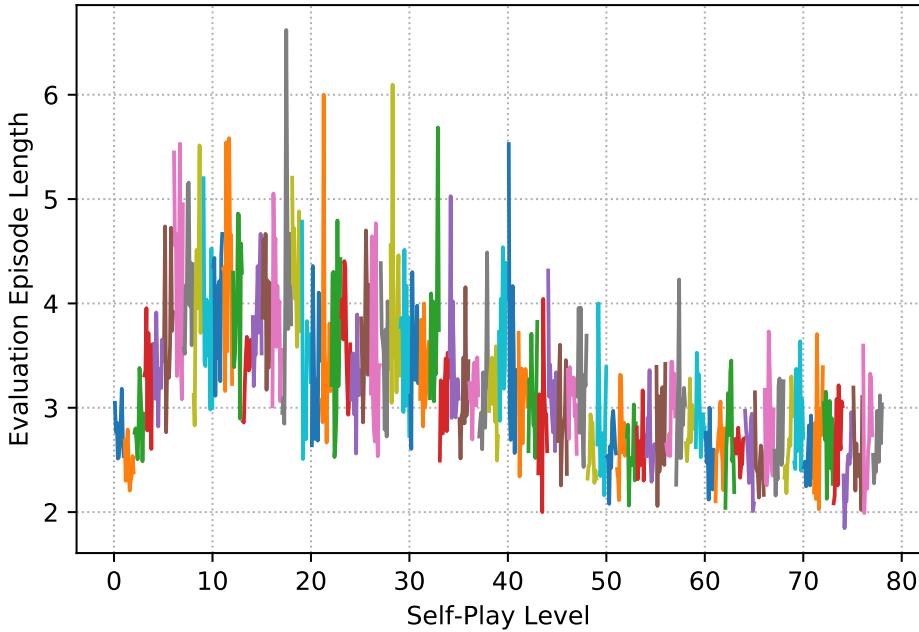
Figure 9.5: **Self-Play Learning of Adversarial Land-Ball Strategy.** Mean evaluation episode length over 77 self-play levels. Each colored line segment corresponds to a different level. The vertical axis shows the average length of an adversarial episode in evaluation. Each point is an average over 240 evaluation episodes. The maximum episode length is 10. As the training process continues, the agent learns to win the point more quickly. The mean episode length reaches four at the start of the training process, and then goes down to about 2.5 after 77 levels.

section evaluates the impact of using the ball-trajectory prediction model by training a strategy over a striking policy that does not use any models.

9.5 Paddle-Control Strategy

As described in Chapter 4, there are three variants of the policy hierarchy with three different striking skills. The previous two sections covered the land-ball and hit-ball striking skills. The third is to use the paddle-control skill directly. The difference be-
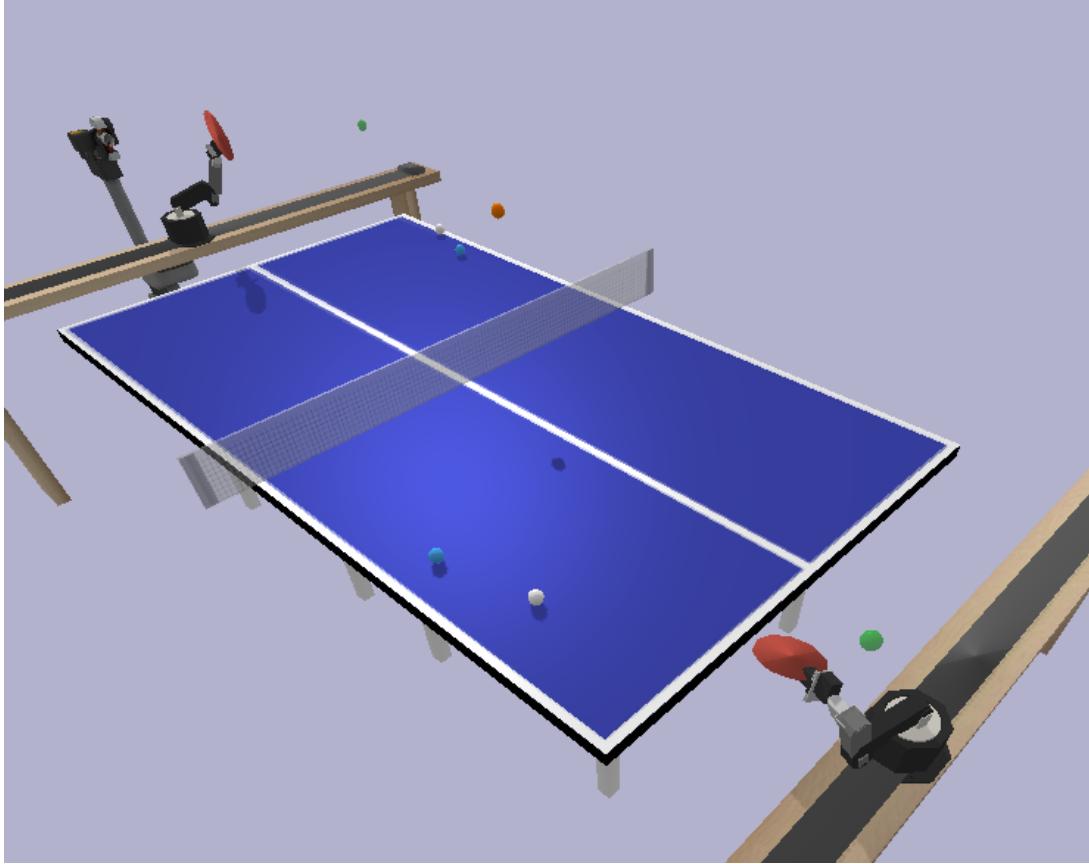
Figure 9.6: **Visualization of Adversarial Land-Ball Strategy.** The agent learns to use fast balls or aim for targets near the edges of the table. In the example shown both robots have aimed for such targets. Landing targets are indicated by white marker balls.

tween this striking policy and the hit-ball policy is that the agent is expected to specify all attributes of the paddle-motion state, including its target position $l(p_t)$ and the corresponding time t. In other words, without access to the ball-trajectory prediction model in the hit-ball skill, this variant is expected to predict a position and time for incoming ball implicitly and aim for that spot with the paddle.

Fig. 9.12 shows the training progress for a cooperative paddle-control strategy. After about 250 self-play levels, the mean episode length hardly reaches two. The hit-ball strategy maxes out the episode length after this number of self-play levels. This experiment was terminated early, but an earlier experiment that ran for many more self-
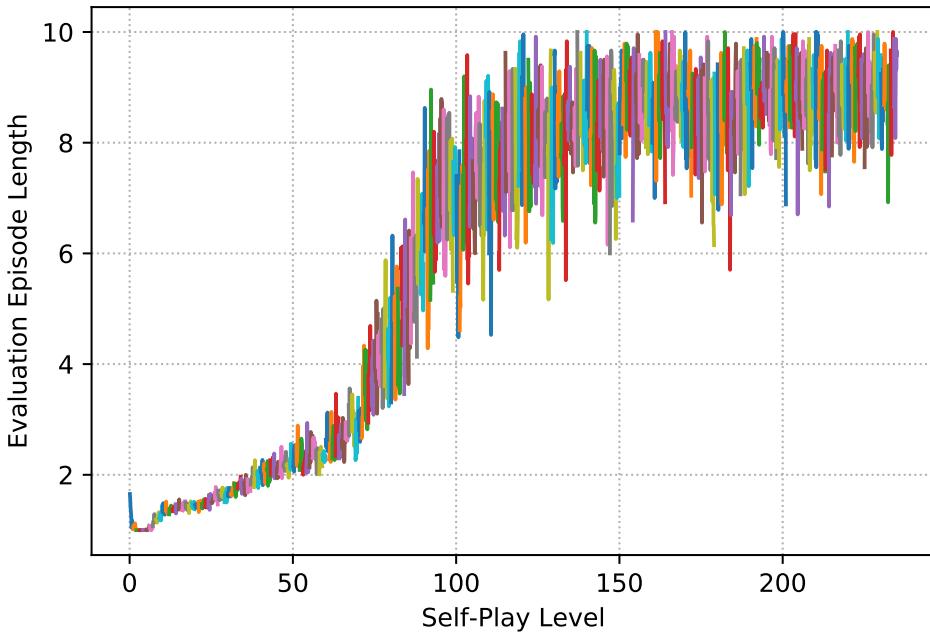
Figure 9.7: **Self-Play Learning of Cooperative Hit-Ball Strategy.** Mean evaluation episode length over 233 self-play levels. Each colored line segment corresponds to a different level. The vertical axis shows the average length of a cooperative episode in evaluation. Each point is an average over 240 evaluation episodes. The maximum episode length is 10. The training progress for the hit-ball strategy is much slower than the land-ball strategy, however, given enough training the hit-ball strategy can discover novel strikes that allow it to score better than the land-ball strategy.

play levels shows that given more time the model-free PPO algorithm is able to find better paddle-control strategies. Fig. 9.13 shows the training progress for an experiment with lasted about 700 self-play levels, corresponding to about 1.7 M training exchanges. In this experiment, the episode cap was set to 100, instead of 10. The strategy agent was able to achieve a mean episode length of about 40 and the performance continues to climb.

Fig. 9.12 shows the training progress for an adversarial paddle-control strategy. The mean episode length hardly rises above one, as was the case with the adversarial hit-ball policy as well. However, with the hit-ball policy, the episodes ended quickly
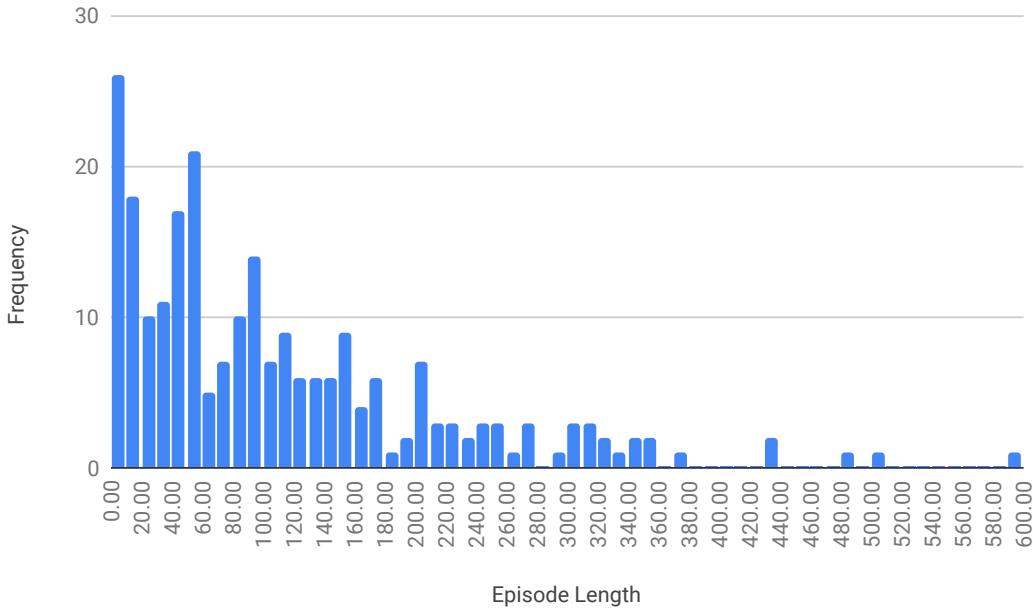
Figure 9.8: **Histogram of Episode Lengths for Cooperative Hit-Ball Strategy.**
Distribution of episode length over 240 test episodes with an episode length cap of 1000.
The hit-ball strategy achieves a mean episode length of 111.9 with a standard deviation of
105.9. It significantly outperforms the land-ball strategy. However, training the hit-ball
strategy requires far more samples than the land-ball strategy.

because after some amount of training either robot could win the rally in one shot. The
paddle-control strategy has a hard time learning to make contact with the ball, so, the
side that has to act first has a high chance of losing the rally by failing the return the ball.
Although it is possible that with more training the adversarial paddle-control strategy
could improve, there is no advantage to using the paddle-control strategy over the hit-ball
strategy, as they both have the same action space.

The experiments in this section show that the simple dynamics model that pre-
dicts the ball's future trajectory can improve the effectiveness and sample-efficiency of
cooperative and adversarial strategies dramatically.

115
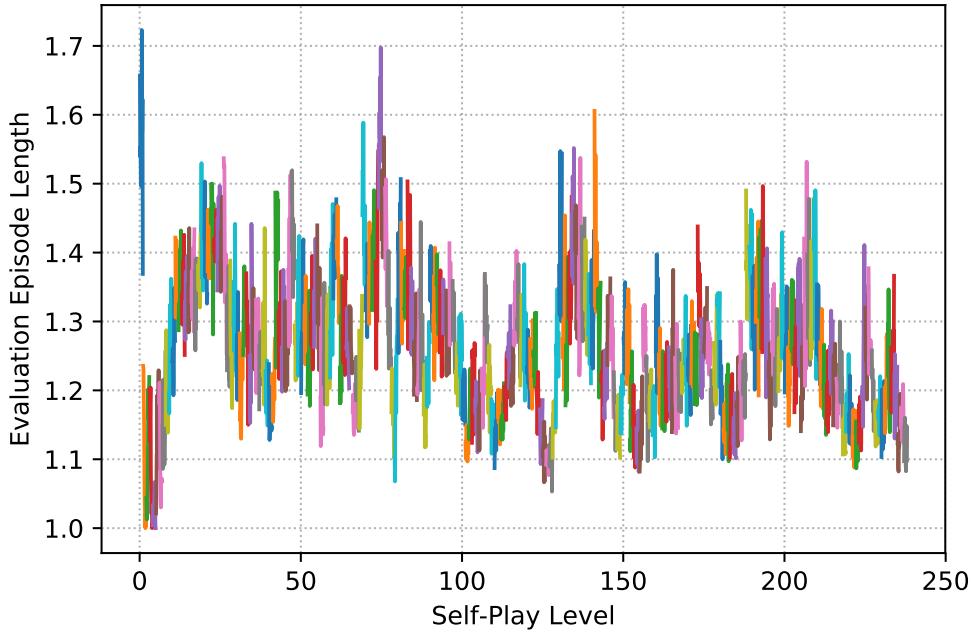
Figure 9.9: **Self-Play Learning of Adversarial Hit-Ball Strategy.** Mean evaluation episode length over 237 self-play levels. Each colored line segment corresponds to a different level. The vertical axis shows the average length of an adversarial episode in evaluation. Each point is an average over 240 evaluation episodes. The maximum episode length is 10. The mean episode length very quickly approaches slightly above one, which signifies that the episodes end very quickly. While at the beginning the episodes end because the robots cannot successfully return the ball, at some point they learn to quickly win the point on the first strike.

9.6   Joint-Control Strategy

Learning a strategy over the joint-control skill requires the agent to specify joint velocity targets on every timestep of the environment. A typical ball exchange lasts 70-100 timesteps. A joint-control strategy needs to execute successful strikes just in time as the ball is reaching the robot, and then properly position the robot to be ready for the next strike. So, in effect, it needs to execute two complementary skills.

Fig. 9.15 shows the results for training an ARS agent to play cooperative games

116
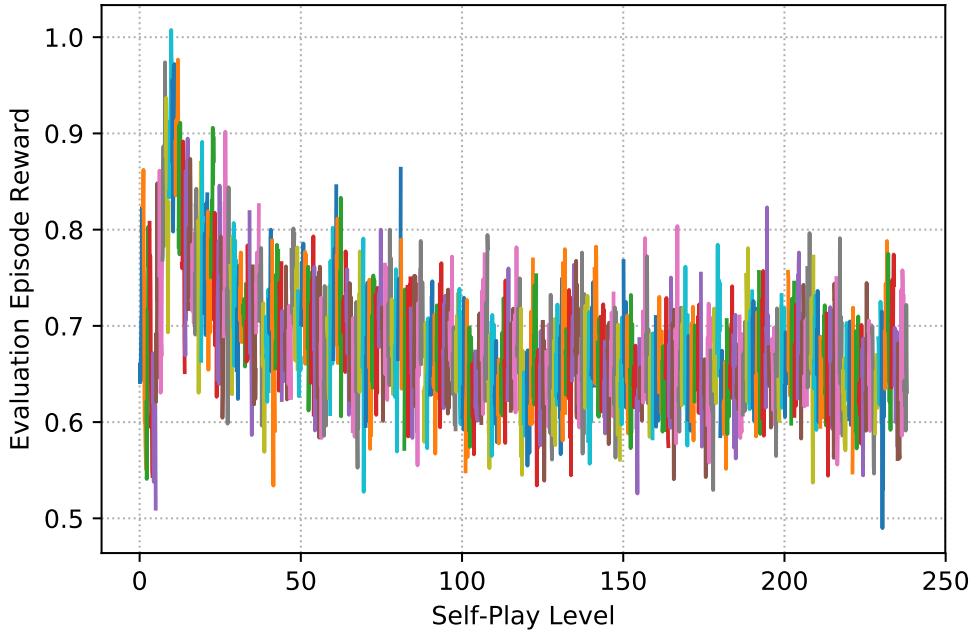
Figure 9.10: **Average Evaluation Rewards for Adversarial Land-Ball Strategy.**
Mean evaluation reward over 237 self-play levels. The mean episode reward reaches close
to one at the beginning, which means the learning agent can easily exploit the opponent
with the frozen policy. With more training, the reward becomes more balanced.

against itself. The ARS policies converge after about 2M episodes. The best policy
achieved a score of 2.0 (both arms return the ball) 87.1% of the time, and a score of 1.0
about 0.1% of the time. While the learned strategy was cooperative in the sense that the
second robot almost always succeeded in returning the ball, this behavior did not extend
to longer plays, i.e. the first robot could not return the ball after the second robot hit it.
Longer cooperative play could possibly be achieved by allowing longer rollouts at training
time, as well as increasing the randomness in the environment initial state (position and
velocity of the ball and robots).

Training the model-free joint-control strategy is more difficult than training the
model-free paddle-control strategy discussed in Sec. 9.5. While the paddle-control strategy
needed many training episodes, it could gradually improve the cooperative rewards and
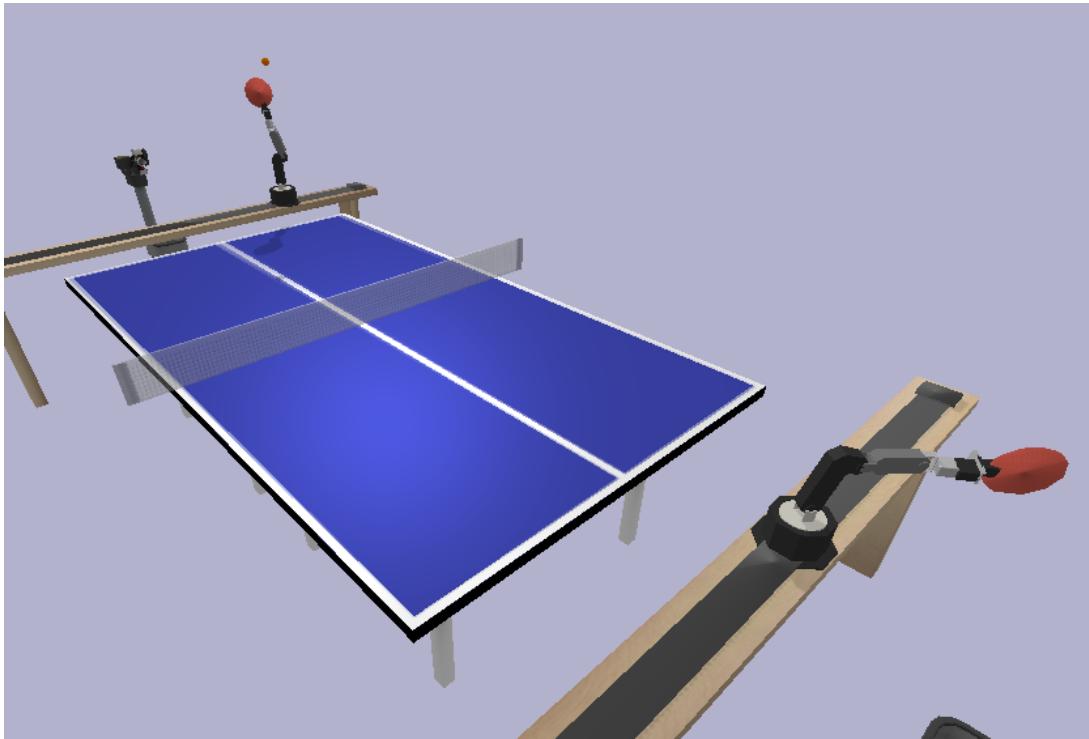
Figure 9.11: **Visualization of Adversarial Hit-Ball Strategy.** The strategy agent learns to win the rally by sending the ball over the opponent.

achieve cooperative episode lengths equal to 40% of the maximum possible after less than 2M training episodes. The rewards achieved by the model-free joint-control policy are lower by comparison. This difference can be attributed to the high-level paddle actions that allow the paddle-control strategy to execute effective paddle strikes just by specifying a target motion state and time for the paddle. The long time horizon of these high-level actions make learning easier by reducing the delay in receiving the reward for the chosen action.

9.7   Conclusion

This chapter discussed training and evaluation of different variants of the strategy skill operating over various striking policies including land-ball, hit-ball, paddle-control, and joint-control. The hierarchical policy design used in this dissertation allows the table-
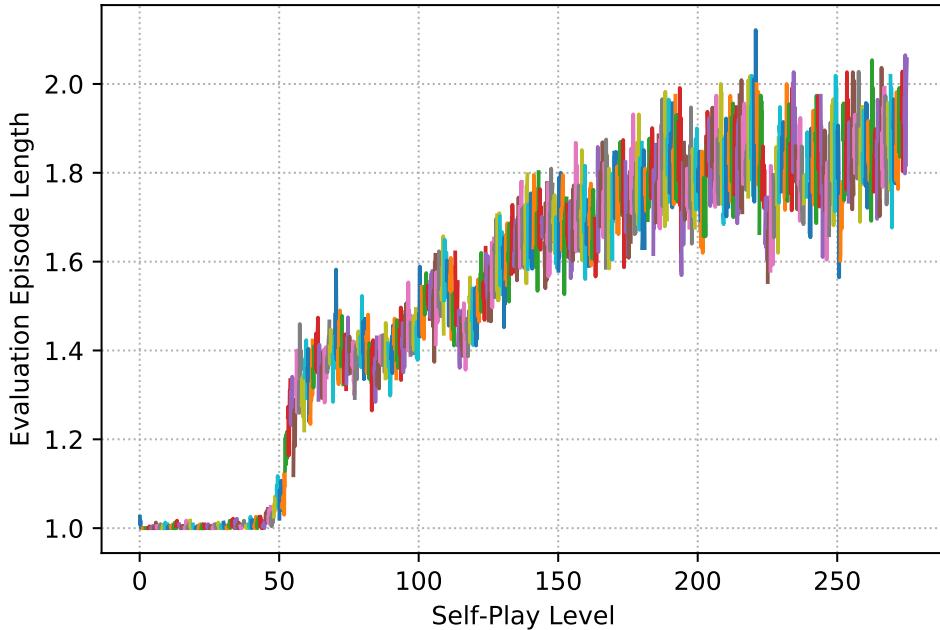
Figure 9.12: **Self-Play Learning of Cooperative Paddle-Control Strategy.** Mean evaluation episode length over 274 self-play levels. Each colored line segment corresponds to a different level. The vertical axis shows the average length of a cooperative episode in evaluation. Each point is an average over 240 evaluation episodes. The maximum episode length is 10. The training process for the paddle-control strategy is much slower than the land-ball and hit-ball strategies, since the agent needs to figure out how to make contact with the ball in the first place.

tennis agent to combine model-based, model-free, and analytic skills in a single policy that can effectively learn cooperative and adversarial games with good sample-efficiency.

Using model-based striking policies greatly increases the sample-efficiency of self-play learning. On the other hand, allowing the strategy skill to execute arbitrary strikes with the hit-ball skill results in discovering novel strikes suitable for cooperative and adversarial games. Self-play strategies over completely model-free skills like paddle-control and joint-control require more more training episode and are less effective at the task.

The experiment results suggest that model-based learning is a good way to in-
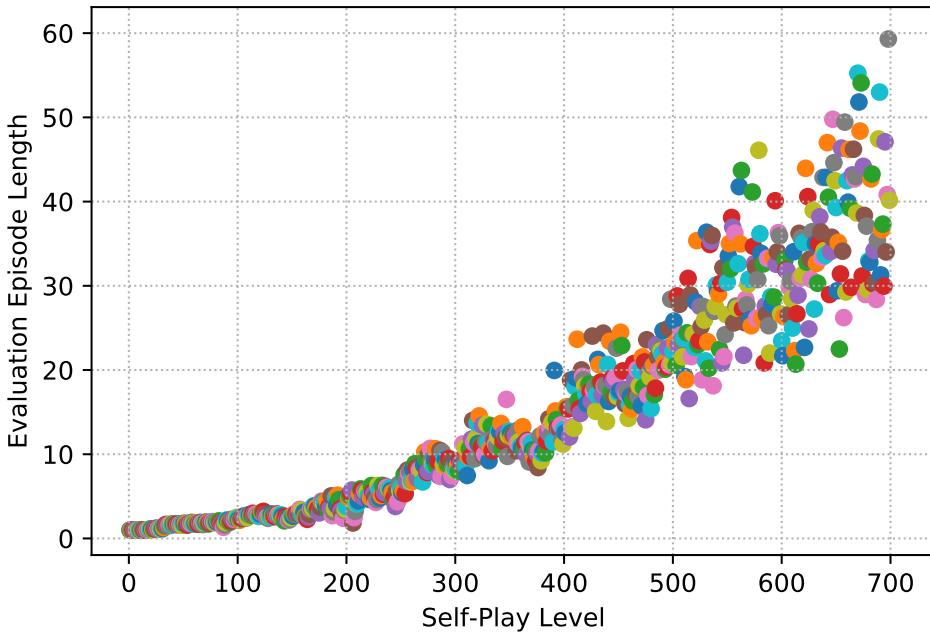
Figure 9.13: **Self-Play Learning of Cooperative Paddle-Control Strategy with More Training Time.** Mean evaluation episode length over 698 self-play levels. The vertical axis shows the average length of a cooperative episode in evaluation. Each colored dot corresponds to a different self-play level and represents the mean over 2400 episodes. Note that unlike the previous cooperative plots, here the maximum episode length is 100. The training process for the paddle-control strategy is much slower than the land-ball and hit-ball strategies, since the agent needs to figure out how to make contact with the ball in the first place.

crease sample-efficiency of learning robotic tasks, especially when the aim is to learn tasks directly in the real world. It is possible to fine-tune the landing models trained from human demonstrations with data collected from other striking policies, as done in the dagger method [23]. Such an approach can combine the sample-efficiency of model-based learning with flexibility of model-free policies.
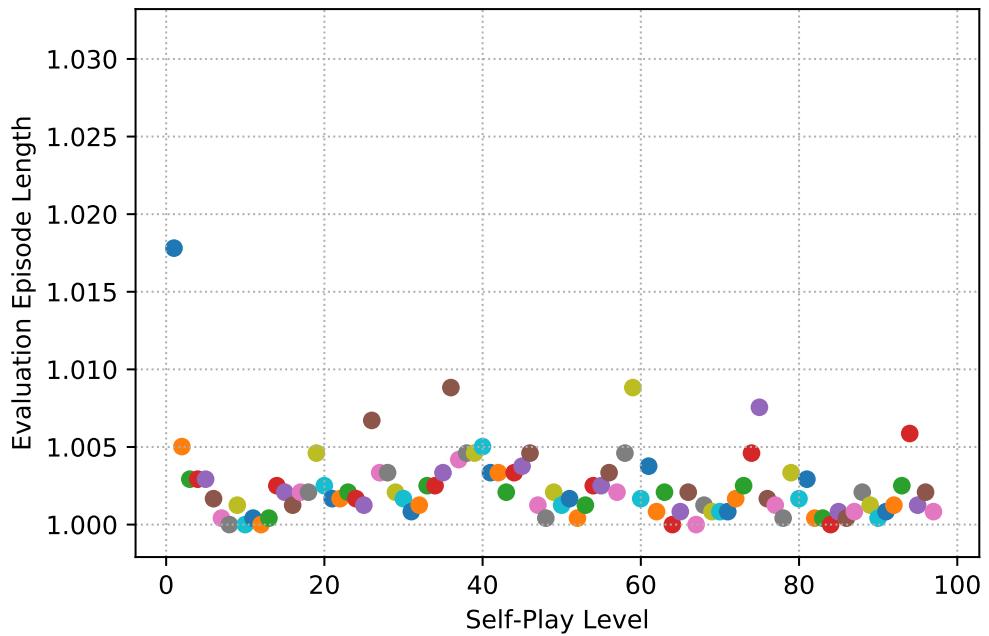
Figure 9.14: **Self-Play Learning of Adversarial Paddle-Control Strategy.** Mean evaluation episode length over 96 self-play levels. The vertical axis shows the average length of a cooperative episode in evaluation. Each colored dot corresponds to a different self-play level and represents the mean over 2400 episodes. The mean episode length hardly goes above one, since neither the learning agent nor the opponent with the frozen policy can learn to make contact with the ball and return it successfully with this much training. In these games, the first robot which gets to act usually loses the point.
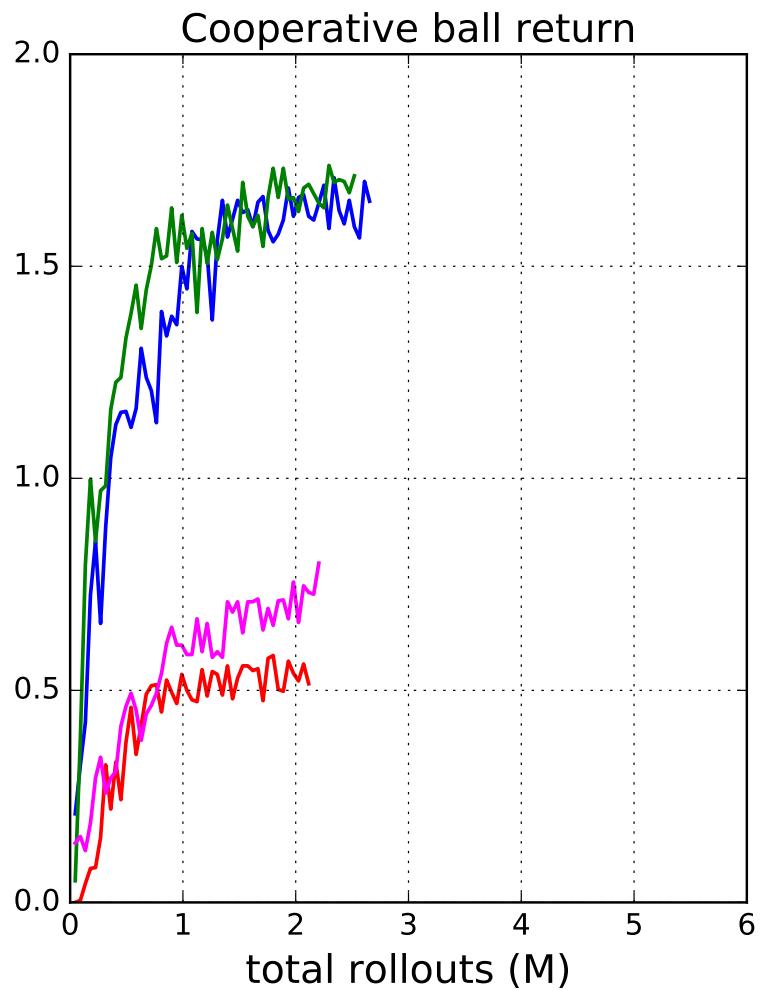
Figure 9.15: **Self-Play Learning of Model-Free Joint-Control Strategy with ARS.** Cooperative scores for four randomly-initialized runs. The vertical axis shows the number of successfully-returned balls per rally. In all four runs the number of evaluations k per policy perturbation was set to 15. Training the model-free joint-control strategy is less effective and less sample-efficient than other strategy skills.

Chapter 10

Discussion and Future Work

This dissertation presents a method for learning the technical and tactical dimensions of robotic table tennis in a virtual reality environment. The technical aspects of the game are learned from realistic human demonstrations, and are executed using a robot-agnostic analytic controller that can execute timed motion targets for the paddle including its pose and linear and angular velocities. The tactical aspect of the game is captured in the strategy skill and trained using model-free RL efficiently. This chapter evaluates the results presented in this dissertation and outlines steps for future work.

10.1 Development Process

The work reported in the literature often focuses on results from the most successful experiments and the best-performing agents. However, it is not always clear how much effort goes into training those agents. In some cases, extensive hyperparameter tuning and experiments with multiple random seeds are needed to train high-performing agents. Such a development process is not feasible in robotic domains, where each training episode carries a significant cost in terms of development time and wear and tear on the hardware.

In an end-to-end setup, the impact of a change in the implementation can be evaluated only by its impact on the overall performance of the agent. Often, learning algorithms are able to function moderately well even with bugs, making it difficult to observe the negative impact of a new bug immediately after it has been introduced. The hierarchical policy used in this dissertation makes the development and debugging of individual skills easier. Since the skills can be evaluated separately, the failure cases and the root causes for performance issues can be identified and studied. Moreover, it is possible to give the policies perfect observations and actions to accurately measure their intrinsic errors.

The high-level skills like striking and positioning in the hierarchical policy give the

top-level strategy skill access to neutral high-level behaviors that it can exploit in different ways to discover new ways of playing table tennis. With this design, the application of model-free RL can be localized to the strategy skill. Since the high-level skills are robust and effective, a greater percentage of the action space of the strategy skill consists of useful actions. So, during exploration, the RL algorithm is less likely to waste its effort on trying actions that are not useful.

Since the underlying skills are robust and action space for the strategy skill is effective, general-purpose RL algorithms like PPO can discover effective policies using only about 24 000 training episodes. All hierarchical reinforcement learning experiments in this dissertation were carried out on one or two local workstations, without hyperparameter tuning. This is a huge improvement over standard practice in model-free RL.

10.2   Driving Different Robot Assemblies

Decomposing the environment into a game space and a robot space makes it possible to learn the game dynamics independently of how robot table tennis is to be played. If the robot assembly is replaced, the game dynamics models can be used with the new robot without retraining.

The analytic paddle controller is also agnostic to the robot setup. Given the motion constraints for the robot, it can optimally drive any robot assembly to execute motion targets for the paddle using the Reflexxes library. The analytic controller treats a multi-robot setup (for example, a linear actuator plus an arm) as a single assembly, which allows it to control complex robot setups.

The action spaces for the striking and positioning skills work with paddle positions. This choice allows the same implementation for these skills to work with different robot assemblies. However, the strategy policies learned on a particular robot assembly would not transfer to a different robot assembly.

Replacing the robot assembly requires updating the action spaces to reflect the area of the space that is reachable by the robot. Also, the canonical forehand and backhand poses that are used to initialize the IK search need to be updated (see Sec. 7.5).

Thus the approach is quite general and independent of the robot, which makes it possible to transfer it to the hardware without much further work on the controller.

10.3 Observation Uncertainty

The simulator used in this dissertation (PyBullet) has no built-in observation and action noise. As discussed in Sec. 10.1 it was beneficial to train the agents with no observation uncertainty to study the best possible performance of the dynamics models and the training policies and gain better insight into any factors that contribute to errors in prediction or behavior. However, deploying this method in the real-world requires working with observation uncertainty.

Even though the underlying simulation environment does not have any observation noise, the method is designed with observation noise in mind. The Reflexxes library was chosen in part because of its ability to react to sensory noise. Reflexxes can plan to reach a target motion state given any current motion state. So, if the paddle target is changed due to updated observations on the trajectory of the ball, Reflexxes can recompute a trajectory to reach a new target from the current motion state.

The land-ball algorithm discussed in Sec. 8.1 is an open-loop algorithm. Working with noisy observations requires implementing closed-loop control policies. Experiments are needed to evaluate and measure any loss of performance due to observation uncertainty. Noisy observations would make the task more difficult. On the other hand, a closed-loop controller can improve performance by continuously updating the predicted trajectory of the incoming ball. In the current implementation, this trajectory is predicted as soon as the ball starts traveling in the direction of the robot. In other words, the striking policy computes – and never updates – a prediction for the position and velocity of the ball when it reaches the robot. This prediction is computed when the ball is on the other side of the table. Closed-loop policies may obtain more accurate predictions when the ball gets closer to the robot. This increase in prediction accuracy can improve the landing accuracy.

Fig. 10.1 shows the environment with observation noise included. At each timestep i, the environment exposes some observations obs_i of the ball. The observations could be the position and velocity estimates from a ball-tracking algorithm, or the raw blob locations for the ball from multiple cameras. The policy needs to estimate the true motion state of the ball from these observations. Such an estimate can be obtained from a model described in Sec. 10.3.1. Since the ball state estimate keeps changing after receiving new observations, Sec. 10.3.2 describes how the striking policies like land-ball can be adjusted to update their predictions and targets during the course of a paddle swing.
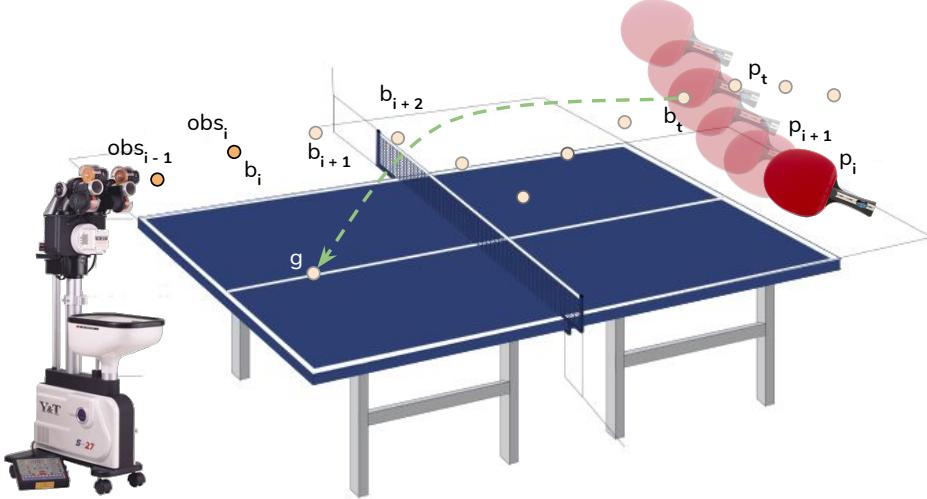
Figure 10.1: **The Game Environment with Observation Noise.** At every timestep i, the environment exposes an observation obs_i of the state of the ball. The observations could be the ball position and velocity estimates coming from a ball tracker, or a set of two-dimensional coordinates for the ball blob from multiple cameras. The current and past observations can be used to estimate the true state of the ball b_i. This setup makes it possible to use the developed approach in environments with observation noise.

10.3.1   Ball-State Estimation Model

The ball-state estimation model receives a sequence of noisy observations of the ball and estimates the ball's current position and velocity. It is described by the function

$$b_0, b_1, \ldots, b_s = S(\text{obs}_0, \text{obs}_1, \ldots, \text{obs}_s), \tag{10.1}$$

where S denotes the ball-state estimation model, obs_s denotes a noisy position observation on the ball obtained at timestep s, and b_s denotes the estimate on the ball's motion state at time s.

Fig. 10.2 shows the network architecture for this model. It is a recurrent model with two LSTM layers followed by a fully-connected layer. At each timestep, the model is
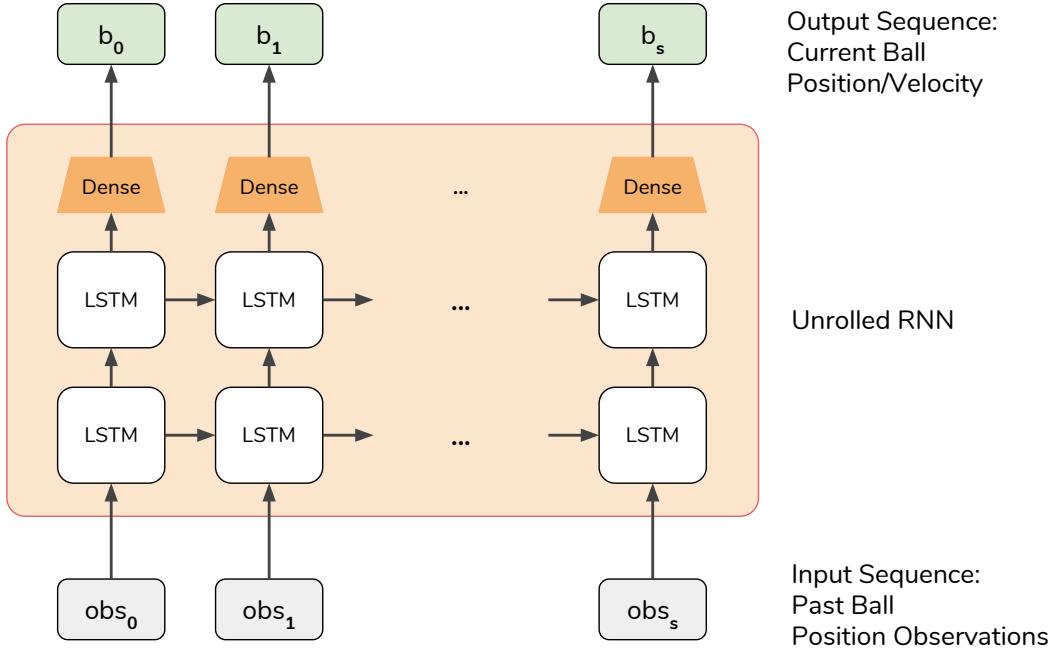
Figure 10.2: **Ball-State Estimation Model.** The current ball-motion state is estimated using a sequence model with LSTM layers followed by a fully-connected layer. The figure shows the model unrolled through time for better visualization. This model makes it possible to estimate the current position and velocity of the ball given a sequence of noisy position observations.

given a new noisy position observation and it produces an updated estimate on the ball's current position and velocity. Over time, with more observations, this model can produce estimates that get closer to the true state of the ball.

The agent maintains one instance of the model. As the episode progresses, on each new timestep, the model is updated with a new observation from the current timestep and produces an updated estimate on the current ball-motion state. The estimate obtained from this model can be used to generate predictions about the future states of the ball using the ball trajectory prediction model as explained in the next section.

10.3.2 Closed-Loop Land-Ball Skill

Algorithm 2 shows the updated land-ball algorithm that works in an environment with noisy observations. Instead of computing a target paddle-motion state p_t only once, this algorithm recomputes the paddle target on every timestep. The algorithm starts by using the ball observations to update its current estimate of the ball-motion state b_s. It then predicts the ball trajectory T, and computes a paddle-motion state p_t to hit the ball toward target g. However, instead of following p_t using an open-loop controller, the algorithm follows p_t just for one timestep. On the next timestep, p_t is updated based on a new ball observation, and the paddle-control skill is requested to adjust its target to the updated p_t. Since Reflexxes is able to adapt to moving targets, the land-ball algorithm works with observation noise as well.

> **inputs :** Sequence of noisy ball observations $\{obs_0, \text{obs}_1, \dots\}$
> **inputs :** Current timestep s
> **inputs :** Desired landing target g
> $i \leftarrow s$
> **repeat**
> > read latest ball observation obs_i
> > $b_i \leftarrow S(\{\{\text{obs}_0, \text{obs}_1, \dots, \text{obs}_i\}\})$
> > $T = b_{i+1}, b_{i+2}, \dots, b_n \leftarrow B(b_i)$
> > **foreach** $b_k \in T$ *such that* b_k *is reachable* **do**
> > > $p_k \leftarrow L^{-1}(b_k, g)$
> > > $\hat{p}_k \leftarrow P(p_k, p_s)$
> > > $\hat{g}_k \leftarrow L(\hat{p}_k, b_k)$
> >
> > **end**
> > $t \leftarrow \arg\min_k \|\hat{g}_k - g\|$
> > emit first action u_i from $\pi_p(t, p_t \mid p_s)$
> > $i \leftarrow i + 1$
>
> **until** *robot paddle hits the ball or episode ends*

Algorithm 2: Closed-Loop Land-Ball Skill Algorithm

Some heuristics are needed to avoid potential failure cases. For example, when the paddle is close to the target, if Reflexxes is moving the paddle with maximum velocity or acceleration, a small update to the target make render it infeasible. In such cases, the update should be ignored, opting for a small error, rather than a failure.

This extension takes a major step towards taking this system to real hardware. Modeling ball spin and using vision to estimate it are other such extensions, which are discussed in subsequent sections.

10.4   Ball Spin and Magnus Forces

The PyBullet simulator does not simulate Magnus forces that cause a spinning object to curve its trajectory in the air. In the current implementation, the ball's spin only affects its motion after contact due to the friction forces acting on the ball.

The method can be extended to factor in the ball's spin. The ball-trajectory prediction model can be extended to receive an estimate on the ball's spin as input. As experiments in Sec. 6.5 show, ball-trajectory prediction is a simple problem and the model is expected to have the capacity to handle ball spin as well. To make it easier to predict the future states of a spinning ball, the dynamics models discussed in Chapter 6 can be augmented with physics models that capture Newtonian motion, air friction, and Magnus forces. The parameters in the physics models can be fitted to observed trajectories. Once fitted, the physics models can be used to establish initial estimates of the future states of the ball. Then the neural network can predict only a residual over the estimates from the physics models, which is a simpler prediction problem. The next section discusses how the ball's spin can be estimated in real-world environments.

10.5   Vision

The method advocates working with low-dimensional state over raw visual input. The primary element in the environment that needs to be observed with vision is the ball. A variety of ball tracking algorithms exist that can compute estimates on the position and velocity of a moving ball. Detecting the ball in a camera image is a relatively simple computer vision task which does not require expensive computations. On the other hand, using neural networks to track the ball using raw visual input requires carefully varying the lighting conditions and backgrounds to avoid overfitting the model to the training environment.

The only situation that requires handling raw visual input is estimating the impact of a human opponent's paddle motion on the ball. Since the opponent paddle is not

instrumented, it would be useful to establish an estimate on its motion at the moment of contact with the ball. Such an estimate can be used to help the ball tracking algorithm predict an expected trajectory for the ball. It can also be used as an input to the ball-state estimation model shown in Fig. 10.2.

Establishing a prior on the motion of the ball as a result of contact with the opponent paddle is specially useful for estimating the ball's spin. While the ball position and velocity can be estimated using a few observations, the ball's spin is harder to estimate. However, during the game the ball always starts with no spin in the player's hand. Physics models can be used to establish an estimate on the ball spin as a result of contact with the human or robot paddles. With these extensions, the method can be deployed in the real world.

10.6 Hardware Evaluation

Experiments in Chapter 8 show that only about 7 000 human demonstrations can be used to train a land-ball policy that can hit ball targets with about 20 cm error. Similarly, experiments in Chapter 9 show that only about 24 000 self-play exchanges on the robot can be used to learn a strategy that can sustain rallies lasting 14-16 hits. So, these experiment suggest that the method may work in the real world as well.

The virtual reality environment developed in this dissertation has many similarities with a real-world table-tennis environment. Most notably, the same motion sensors used for tracking the paddle in the VR environment can be used to track human paddles in the real world. The sensory readings from VR trackers are similar to readings from any other motion tracking system. The data collection setup used in this dissertation already deals with many problems arising from working with physical sensors. For example, there is often jitter present in location readings coming from the VR tracker, which makes estimating the true position of the VR paddle difficult. Also, the VR trackers do not provide any velocity information. The data collection program estimates the linear and angular velocity from the position readings. Lastly, HTC VR trackers work at 90 Hz, while the simulator works at 1000 Hz. The difference in frequency requires producing intermediate estimates for the state of the tracker.

The VR environment can be regarded as an implementation between a simulation and the real world. Although the sensory readings coming from VR trackers are similar to sensory readings that would be available in the real world from a motion tracking

system, deploying the method in the real world requires obtaining the ball state from vision using a ball-tracking algorithm. As outlined in Sec. 10.3, the striking policies need to be updated to work in a closed-loop manner and update their targets based on new ball observations. Additionally, deploying the method in the real world requires training a residual dynamics model for the paddle-control policy to capture control inaccuracies due to imperfect robot hardware and robot controllers.

The experiments carried out in this dissertation show that a robotic table-tennis agent can be trained using only a few thousand human demonstrations and about 20 thousand exchanges in robot games. This high sample-efficiency suggests that the method can work in the real world as well. It is likely that training the models and policies in the real world requires dealing with additional challenges that would reduce the sample-efficiency or performance of the method. However, there are also opportunities for increasing the sample-efficiency or performance, e.g. by augmenting demonstrated trajectories using height reduction (Chapter 6), augmenting dynamics models with physics models (Sec. 6.1.2), using closed-loop controllers (Sec. 10.3.2), and mixing dynamics models with model-free policies as discussed in the next section.

10.7   Mixing Dynamics Models with Model-Free Policies

Dynamics models greatly improve the sample-efficiency of the agent. Only about 7 000 demonstrated strikes were enough to train a striking policy with a mean target error of about 20 cm. Similarly, only about 24 000 self-play exchanges were enough to train a cooperative land-ball strategy sustaining rallies lasting about eight exchanges on average. These results show that model-based learning requires much fewer samples. On the other hand, the dynamics models that were trained from human demonstrations were never updated during robot experiments. Therefore, the land-ball striking policy that uses these dynamics models stays limited to strikes that were demonstrated by humans.

The hit-ball strategy experiments in Sec. 9.4 show that more flexible striking skills can discover striking motions beyond what is observed in demonstrations. On the other hand, since the hit-ball policy uses only one dynamics model for ball-trajectory prediction, and its action space has more dimensions compared to the entirely model-based land-ball skill, training a strategy over hit-ball is less sample-efficient.

The method can be extended to allow updating the dynamics models with data collected from more flexible policies that are either model-free or use fewer dynamics

131

models. Doing so would increases the predictive ability of the dynamics models to cover a wider range of behaviors in the environment.

On the other hand, rather than learning the skills from scratch, the model-free policies can be modified to use the predictions from the models as a starting point for their decisions. Doing so allows the model-free policies to expand the boundary of explored behaviors gradually by trying out variations slightly outside the space of behaviors captured in the models. Such guided exploration would make of the model-free policies more sample-efficient.

10.8   Conclusion

This dissertation demonstrated sample-efficient learning of technical and tactical aspects of robotic table tennis. Since the experiments were carried out in a VR environment with real-world motion sensors, the results suggest that the method can be used to learn the task in the real world without relying on transfer of models or policies from simulation. The ability of Reflexxes to adjust the trajectory target would dynamically allow the method to work with noisy ball position and velocity estimates coming from a ball-tracker in the real world.

Chapter 11

Related Work

11.1 Robot Table Tennis

There has been a broad body of work on robot table tennis [1, 2, 15]. More recent approaches by Muelling et al. [20, 17, 19, 18, 32] are similar to the approach in this dissertation in that they use human demonstrations to seed the robot's behavior and they decompose the task into high-level and low-level skills.

Inspired by studies of humans playing table tennis, Muelling et al. [20] break up the game into four different stages – awaiting, preparation, hitting, and finishing. In the hitting stage, they use an inverse dynamics model to compute the parameters of the robot at time of contact to achieve a desired target location, similar to the land-ball skill presented in this dissertation. Muelling et al. also decompose the task of playing table tennis into subtasks, which is similar to this work. They implement the low-level control of the robot by learning a set of hitting movements from human-guided kinesthetic demonstrations on the robot [19]. These movements are compiled into a library of motor primitives, which are then expanded by mixing existing movements to create new motor primitives. They then employ inverse reinforcement learning (IRL) over human demonstrations to learn a high-level strategy for the full game [18].

Rather than using kinesthetic teaching, the approach in this dissertation captures human strikes using instrumented paddles in free-form human games. This choice avoids limiting the human's movements to what is possible with kinesthetic teaching. Instead of representing hitting movements at the joint level, the method in this dissertation abstracts strikes by the paddle-motion state that is in effect at the time of contact. This high-level representation makes it possible to solve the motor control problem using the general-purpose Reflexxes [10] trajectory planning algorithm. Since the paddle-motion state is not coupled to any specific robot, this approach can execute the learned strikes on any robot without a need for new demonstrations.

Muelling et al. [18] use human demonstrations to learn a reward function for the game, which represents a high-level game-play strategy. The method in this dissertation uses human demonstrations only to learn the game dynamics, which gives the policy the freedom to discover more diverse game-playing strategies beyond what is demonstrated by humans. Using self-play in robot vs. robot games makes it possible to to freely explore and refine game-play strategies. Also, the method proposed in this work is hierarchical and assembles both high-level and low-level skills into one overall system, which can be refined end-to-end if needed. This is while Muelling et al. [17] train the different components of the hierarchy as separate models.

11.2 Model-Based and Model-Free Learning

The approach in this dissertation decomposes the problem into parts that are model-based, such as dynamics models of the ball motion, and of contact, and parts that are model-free, such as learning a high-level strategy. This decomposition breaks down the problem in a way that is consistent with the strengths of these different approaches. The model-based parts are trained with supervised models and it is easy to generate data for them. Meanwhile, using model-free techniques for training the strategy allows much better exploration without being very expensive, since only a few dimensions are optimized in the strategy. The work by Muelling et al. [17, 20] follows a similar decomposition overall, but high-level strategies are not trained with model-free approaches. Instead, the strategy is either hard-coded [17] or learned from demonstrations using inverse reinforcement learning techniques [20]. In this dissertation, the high sample-efficiency makes it possible to employ model-free techniques for higher-level skills, which in turn leads to discovering novel strikes and game-play strategies.

11.3 Hierarchical Reinforcement Learning

Hierarchical reinforcement learning is a long-established discipline. From the original options framework [28, 22] to the more recent approaches, like FeuDAL networks [6, 30, 21], there is a broad body of work on training layered policies where each skill sets goals for other skills below it. These approaches share the same underlying goal – namely, to learn policies with a reduced number of timesteps by using models that operate at different rates. In FeuDAL networks the policy consists of two task levels that are

trained together, such that the higher-level task receives rewards over the actual objective in the environment, while the lower-level task receives rewards for achieving targets set by the higher-level task. A common motivation for hierarchical reinforcement learning methods is automated discovery of useful subtasks. However, in this dissertation the main motivation is developing a specific task hierarchy that is most suitable for learning robotic tasks with extremely high sample-efficiency. In this work, the targets set by the higher-level skills for the lower-level skills have specific semantics; they either signify a desired outcome in the environment, such as a ball target location, or indicate a desired state for the robot, such as a target paddle-motion state at the time of contact. Such intermediate targets are effective at reducing the dimensionality of the control problem while allowing for incorporating human demonstrations in a robot-agnostic controller.

11.4 Self-Play Learning

Self-play strategies have been applied extensively in fully-observable games such as Go and chess [27]. Lately, these have also been applied to partially-observed games such as StarCraft [31]. Self-play has not previously been applied to continuous-control tasks such as robotic table tennis, mainly because existing self-play techniques require a prohibitive number of training episodes, which is not feasible for robotic domains. The hierarchical policy and the action representations developed in this work greatly reduce the dimensionality of the higher-level skills like the strategy skill, thereby reducing the number of self-play episodes needed. Effective application of self-play learning makes it possible to learn more diverse striking motions and cooperative and adversarial table-tennis strategies than would be possible through learning from demonstrations or inverse reinforcement learning.

11.5 Underlying Methods

The following libraries and algorithms are used in developing the method in this work.

11.5.1 Reflexxes

Reflexxes [10] is an online trajectory planning algorithm that is originally designed to allow robots to react to unforeseen events. It is capable of computing robot motion trajectories from arbitrary initial states of motion to reach a target state of motion while satisfying given motion constraints. Fig. 11.1 by Kröger [10] illustrates the interface to Reflexxes. Reflexxes is able to produce time-optimal trajectories that take the robot from a given current state of motion to a desired target state of motion without violating the set of given motion constraints.
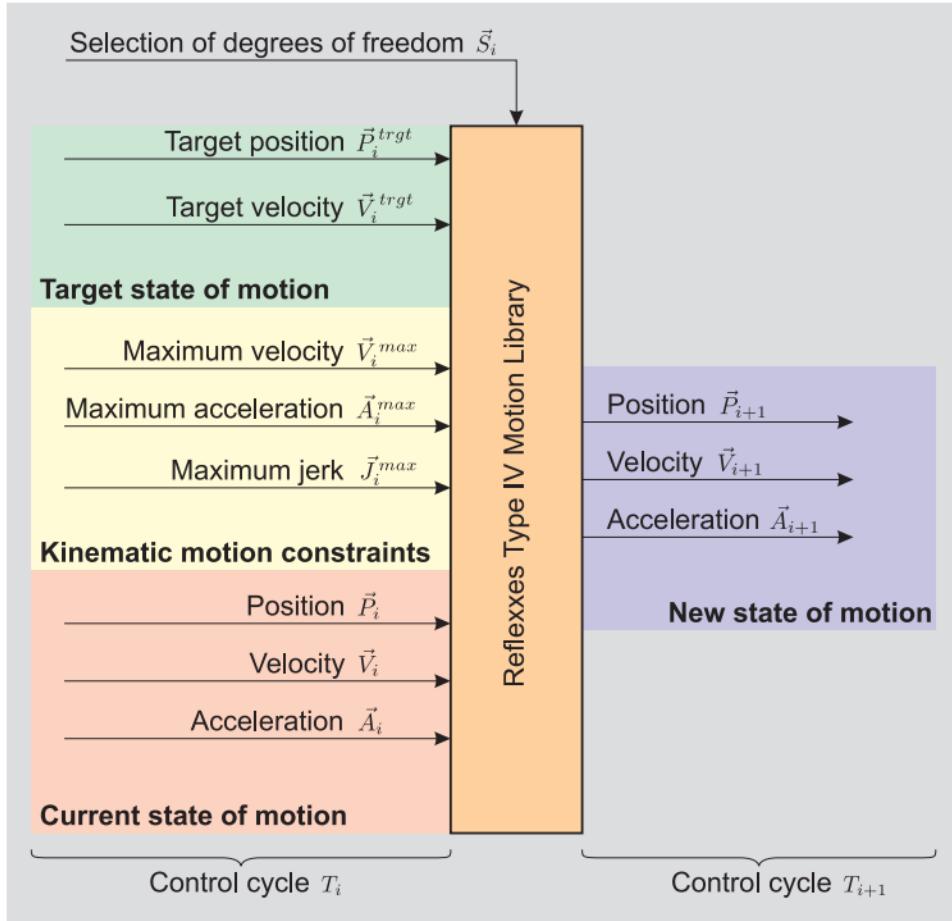
Figure 11.1: **The Interface to Reflexxes Motion Libraries.** At each control cycle, Reflexxes receives the current state of motion (including position, velocity, and acceleration), the target state of motion (including position and velocity), and kinematic motions constraints (including velocity, acceleration, and jerk). Given these inputs, Reflexxes produces the new state of motion (including position, velocity, and acceleration) for the next control cycle. The new state of motion is computed using a fast deterministic algorithm, and is guaranteed to be on a time-optimal trajectory toward the target state of motion. Image source: Kröger[10].

Reflexxes has been used [12] to smooth out the end-effector targets continuously produced by a neural network so that robot movements do not violate velocity, acceleration, and jerk limits. In this work, the motion targets for Reflexxes also act as a

temporally-abstract action representation, which helps reduce the dimensionality of the control problem.

11.5.2 Proximal Policy Optimization (PPO)

Proximal policy optimization [25] is a policy gradient method for model-free reinforcement learning which optimizes a surrogate objective function with respect to the parameters θ of a policy $\pi_\theta(a|s)$ using stochastic gradient ascent. The objective optimized by PPO is

$$L(\theta) = \widehat{\mathbb{E}}\left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} - \beta \mathrm{KL}(\pi_{\theta_{old}}(\cdot|s_t), \pi_\theta(\cdot|s_t))\right] \tag{11.1}$$

where $\widehat{A}_t$ is an estimator of the advantage function at time t, and $\widehat{\mathbb{E}}$ indicates the empirical average over a batch of samples. The simplest version of PPO optimizes 11.1 with $\beta = 0$ and clips the policy ratio $\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$ so as to prevent numerical issues. A different version of PPO adaptively updates β.

In comparison to related policy gradient methods, the objective optimized by PPO is similar to that of trust region policy optimization (TRPO). The main difference is TRPO imposes a hard constraint on the KL divergence between successive policies and finds solutions using the conjugate gradient method. Empirically, both TRPO and PPO have been found to lead to more stable policy updates than the standard policy gradient methods.

11.5.3 Augmented Random Search (ARS)

Random search methods for model-free RL directly optimize the policy by searching over its parameters. The simplest version of random search computes a finite-difference approximation of the gradient along a direction chosen uniformly at random on a sphere centered around the current parameters. Let r_θ be a sample return corresponding to rolling out a trajectory with policy parameters θ, and let δ_k be a random perturbation. Basic random search approximates the gradient as

$$\widehat{g} = \frac{1}{N}\sum_{k=1}^{N}(r_{\theta+\delta_k} - r_{\theta-\delta_k})\delta_k \tag{11.2}$$

The Augmented Random Search algorithm proposed by [14] incorporates several heuristics into the basic search: (1) state vectors are whitened based on an online estimate of state mean and covariance, (2) gradient is estimated based on top b directions with highest return rather than all directions, and (3) the learning rate in each update is scaled by the standard deviation of the obtained rewards. In addition to being simple to describe and implement, ARS has been shown to have competitive performance with other popular model-free algorithms on MuJoCo [29] benchmark environments.

Chapter 12

Conclusion

This chapter enumerates the contributions in this work and concludes the dissertation.

12.1 Contributions

The list below enumerates the contributions in this work.

- Integrating the simulator with a virtual reality environment permits capturing realistic paddle strikes demonstrated by human players. Such strikes contain versatile paddle motions that are not comparable to simple trajectories constructed from smooth splines.

- The hierarchical policy permits mixing model-free, model-based, and analytic policies in a single agent. Decomposing the task into subtasks reduces the number of inputs and outputs for individual subtasks, which makes learning them easier.

- Decomposing the environment into a game space and a robot space permits learning the game dynamics independently of how robot table tennis is to be played. The environment decomposition also permits transferring the game dynamics models to new robots without a need for retraining them.

- Using timed motion targets (including pose and velocity) as high-level actions permits encoding complex robot motions with a simple action representation that is suitable for learning algorithms.

- The robot-agnostic analytic paddle controller can drive any robot assembly to execute paddle strikes specified by timed motion targets for the paddle.

- Normalizing and subsampling trajectories recorded from human demonstrations permits training dynamics models with few samples. Using only about 7000 trajectories, a land-ball policy is trained, which can hit targets with about 20 cm error on average.

- The analytic controller and the model-based land-ball policy can drive a robot assembly to play table-tennis rallies lasting 6-8 strikes on average, without any training on the robot.

- Localizing the application of model-free RL to the strategy skill simplifies the reinforcement learning and exploration problems. Most RL experiments for learning the strategy policies were carried out on two local workstations and without any hyperparameter tuning.

- Learning the strategy skill with self-play permits discovering novel cooperative and adversarial game-plays using the same paddle strikes demonstrated by humans. After only about 24000 ball exchanges in self-play, a game-play strategy is learned that can sustain rallies lasting 14-16 hits on average.

- Applying self-play to more flexible skills like hit-ball results in discovery of novel strikes beyond what was used by humans. These strikes are customized for cooperative or adversarial games.

- Successfully training a functioning table-tennis agent using in the order of tens of thousands human and robot samples demonstrates that the method is sample-efficient enough that it may be deployed in the real world to learn table tennis on physical robots without relying on transfer of models and policies from simulation.

12.2   Conclusion

The intelligent household robots of the coming decades need to be able to learn by observing humans. They also need to be able to figure out how to complete a task using their knowledge of how the world and its objects work. If they have to experiment with a task, they should be able to learn a lot from a few tries.

This dissertation takes a step in that direction by demonstrating the possibility of learning a complex robotic task in a sample-efficient way. The hierarchical policy

design allows for incorporating knowledge of the world by observing humans completing the same task, without restricting the agent to the behavior demonstrated by humans. Employing simple yet highly-expressive action representations and analytic controllers in the underlying skills gives the higher-level skills the freedom to explore the space of high-level behaviors efficiently, leading to accelerated discovery of novel behaviors that are perceived as *intelligent* by human observers.

Bibliography

[1] Russell L Andersson. *A Robot Ping-pong Player: Experiment in Real-time.* MIT Press, 1988.

[2] L Ángel, JM Sebastián, R Saltarén, R Aracil, and R Gutiérrez. Robotenis: design, dynamic modeling and preliminary control. In *Advanced Intelligent Mechatronics. Proceedings, 2005 IEEE/ASME International Conference on*, pages 747–752. IEEE, 2005.

[3] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

[4] Hua-Tsung Chen, Wen-Jiin Tsai, Suh-Yin Lee, and Jen-Yu Yu. Ball tracking and 3d trajectory approximation with applications to tactics analysis from single-camera volleyball sequences. *Multimedia Tools and Applications*, 60(3):641–667, 2012.

[5] Erwin Coumans. Bullet physics engine. *Open Source Software: http://bulletphysics.org*, 2010.

[6] Peter Dayan and Geoffrey E Hinton. Feudal reinforcement learning. In *Advances in neural information processing systems*, pages 271–278, 1993.

[7] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pages 248–255. Ieee, 2009.

[8] Danijar Hafner, James Davidson, and Vincent Vanhoucke. Tensorflow agents: Efficient batched reinforcement learning in tensorflow. *arXiv preprint arXiv:1709.02878*, 2017.

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

[10] Torsten Kröger. Opening the door to new sensor-based robot applications—the reflexxes motion libraries. In *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pages 1–4. IEEE, 2011.

[11] David G Lowe. Object recognition from local scale-invariant features. In *Computer vision, 1999. The proceedings of the seventh IEEE international conference on*, volume 2, pages 1150–1157. Ieee, 1999.

[12] Reza Mahjourian. *Neuroevolutionary Planning for Robotic Control*. University of Texas at Austin, 2016.

[13] Reza Mahjourian, Risto Miikkulainen, Nevena Lazic, Sergey Levine, and Navdeep Jaitly. Hierarchical policy design for sample-efficient learning of robot table tennis through self-play. *arXiv preprint arXiv:1811.12927*, 2018.

[14] Horia Mania, Aurelia Guy, and Benjamin Recht. Simple random search provides a competitive approach to reinforcement learning. *arXiv preprint arXiv:1803.07055*, 2018.

[15] Fumio Miyazaki, Michiya Matsushima, and Masahiro Takeuchi. Learning to dynamically manipulate: A table tennis robot controls a ball and rallies with a human being. In *Advances in Robot Control*, pages 317–341. Springer, 2006.

[16] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[17] Katharina Muelling. *Modeling and learning of complex motor tasks: A case study with robot table tennis*. PhD thesis, Technische Universität, 2013.

[18] Katharina Muelling, Abdeslam Boularias, Betty Mohler, Bernhard Schölkopf, and Jan Peters. Learning strategies in table tennis using inverse reinforcement learning. *Biological cybernetics*, 108(5):603–619, 2014.

[19] Katharina Mülling, Jens Kober, Oliver Kroemer, and Jan Peters. Learning to select and generalize striking movements in robot table tennis. *The International Journal of Robotics Research*, 32(3):263–279, 2013.

[20] Katharina Mülling, Jens Kober, and Jan Peters. A biomimetic approach to robot table tennis. *Adaptive Behavior*, 19(5):359–376, 2011.

[21] Ofir Nachum, Shane Gu, Honglak Lee, and Sergey Levine. Data-efficient hierarchical reinforcement learning. *arXiv preprint arXiv:1805.08296*, 2018.

[22] Doina Precup. *Temporal abstraction in reinforcement learning*. University of Massachusetts Amherst, 2000.

[23] Stéphane Ross, Geoffrey Gordon, and Drew Bagnell. A reduction of imitation learning and structured prediction to no-regret online learning. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 627–635, 2011.

[24] Reuven Rubinstein. The cross-entropy method for combinatorial and continuous optimization. *Methodology and computing in applied probability*, 1(2):127–190, 1999.

[25] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[26] Yongduek Seo, Sunghoon Choi, Hyunwoo Kim, and Ki-Sang Hong. Where are the ball and players? soccer game analysis with color-based tracking and image mosaick. In *International Conference on Image Analysis and Processing*, pages 196–203. Springer, 1997.

[27] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354, 2017.

[28] Richard S Sutton, Doina Precup, and Satinder Singh. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial intelligence*, 112(1-2):181–211, 1999.

[29] Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*, pages 5026–5033. IEEE, 2012.

[30] Alexander Sasha Vezhnevets, Simon Osindero, Tom Schaul, Nicolas Heess, Max Jaderberg, David Silver, and Koray Kavukcuoglu. Feudal networks for hierarchical reinforcement learning. *arXiv preprint arXiv:1703.01161*, 2017.

[31] Oriol Vinyals, Timo Ewalds, Sergey Bartunov, Petko Georgiev, Alexander Sasha Vezhnevets, Michelle Yeo, Alireza Makhzani, Heinrich Küttler, John Agapiou, Julian Schrittwieser, et al. Starcraft ii: A new challenge for reinforcement learning. *arXiv preprint arXiv:1708.04782*, 2017.

[32] Zhikun Wang, Abdeslam Boularias, Katharina Mülling, Bernhard Schölkopf, and Jan Peters. Anticipatory action selection for human–robot table tennis. *Artificial Intelligence*, 247:399–414, 2017.