NeuroAnimator: Fast Neural Network Emulation and Control of Physics-Based Models

by

Radek Grzeszczuk

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto

Abstract

NeuroAnimator: Fast Neural Network Emulation and Control of Physics-Based Models

Radek Grzeszczuk
Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto
1998

Animation through the numerical simulation of physics-based graphics models offers unsurpassed realism, but it can be computationally demanding. Likewise, finding controllers that enable physics-based models to produce desired animations usually entails formidable computational cost. This paper demonstrates the possibility of replacing the numerical simulation and control of model dynamics with a dramatically more efficient alternative. In particular, we propose the NeuroAnimator, a novel approach to creating physically realistic animation that exploits neural networks. NeuroAnimators are automatically trained off-line to emulate physical dynamics through the observation of physics-based models in action. Depending on the model, its neural network emulator can yield physically realistic animation one or two orders of magnitude faster than conventional numerical simulation. Furthermore, by exploiting the network structure of the NeuroAnimator, we introduce a fast algorithm for learning controllers that enables either physics-based models or their neural network emulators to synthesize motions satisfying prescribed animation goals. We demonstrate NeuroAnimators for passive and active (actuated) rigid body, articulated, and deformable physics-based models.

Acknowledgements

First I would like to thank my advisor Demetri Terzopoulos and my *de facto* co-advisor Geoffrey Hinton. This thesis would not have been possible without their support.

I am most grateful to Demetri for his wonderful teaching philosophy. He believes that a student achieves the best results if he or she enjoys the work. Consequently, he gives his students total freedom to pursue their interests. It is not surprising that so many of them become great scientists.

I would also like to thank Demetri for his unbounded enthusiasm, great expertise, resourcefulness, and patience. I could never quite understand how he can always remain so accessible and friendly, despite being the busiest person I know.

I am indebted to Geoffrey for playing such an important role in this thesis. It has been a great pleasure working with him. His knowledge, enthusiasm, and sense of humor are unparalleled. He is a great teacher, capable of presenting very complicated concepts with ease and clarity.

My special thanks go to John Platt for generously agreeing to serve as the external examiner of my thesis, despite his busy schedule. Special thanks go to Professor Anastasios Venetsanopoulos for serving as the internal examiner. Many thanks to Professors Eugene Fiume and Michiel van de Panne for serving on my committee and providing valuable comments on my thesis.

I would like to acknowledge Jessica Hodgins for enabling me to perform the experiments with the human runner model by supplying the SD/FAST mechanical model, the model simulation data for network training, and the display software.

I thank Zoubin Ghahramani for valuable discussions that led to the idea of the rotation and translation invariant emulator, which was crucial to the success of this work. I am indebted to Steve Hunt for procuring the equipment that I needed to carry out our research at Intel. Sonja Jeter assisted me with the Viewpoint models and Mike Gendimenico for set up the video editing suite and helped me to use it. I thank John Funge and Michiel van de Panne for their assistance in producing animations, Mike Revow and Drew van Camp for assistance with Xerion, and Alexander Reshetov for his valuable suggestions about building physical models.

I would also like to acknowledge the wonderful people of the graphics and vision labs at the University of Toronto, and Kathy Yen for helping me to arrange the defense.

I am most grateful to my parents for their support and encouragement, and to Kasia for her love and patience.

Contents

List of Figures

Chapter 1

Introduction

Computer graphics studies the principles of digital image synthesis (Foley et al., 1990). As a field it enjoys enormous popularity commensurate with the visual versatility of its medium. The computer, on the one hand, can deliver hauntingly unusual forms of visual expression while, on the other hand, it can produce images so real that the only trait suggesting their electronic identity is that they may look too perfect.

This dissertation is concerned with computer animation, an important area of computer graphics which focuses on techniques for synthesizing digital image sequences of graphics objects whose forms and motions are modeled mathematically in the computer. Currently, the most popular method in commercial computer animation is *keyframing*—a technique that has been adopted from traditional animation (Lasseter, 1987). In keyframing, a human animator interactively positions graphics objects at key instants in time. The computer then interpolates these so-called "keyframes" to produce a continuous animation. In principle, keyframing affords the animator a high degree of flexibility and direct control in specifying desired motions. In practice, however, keyframing complex graphics models usually requires a great deal of skill and intense labor on the part of the animator, especially when the task is to create physically realistic animation in three dimensions. Hence, graphics researchers are motivated to develop animation techniques that afford a higher degree of automation than is available through keyframing.

In this thesis, we are mainly interested in the process of synthesizing realistic animation in a highly automated fashion. Physical modeling is an important approach to this end. Physics-based computer animation applies the principles of Newtonian mechanics to achieve physically realistic motions. In this paradigm, the animator specifies the physical properties of the graphics model and the equations that govern its motion, and the computer synthesizes the animation through numerical simulation of the governing equations. Once the simulation is initialized, the outcome depends entirely on the forces acting on the physical model. Physics-based animation offers a high degree of realism and automation; however, the animator can exercise only indirect control over the animation, by applying appropriate control forces to the evolving model. The "physics-based animation control problem" is the problem of computing the control forces such that the dynamic model produces motions that satisfy the goals specified by the animator.

After more than a decade of development, physics-based animation techniques are beginning to find their way into high-end commercial animation systems. However, a well-known drawback has retarded their broader penetration—compared to purely geometric models, physical models typically entail formidable numerical simulation costs and they are difficult to control. Despite dramatic increases in the processing power of computers in recent years, physics-based models are still far too expensive for general use. Physics-based models have been underutilized in virtual reality and computer game applications, where interactive display speeds are necessary. Our goal is to help make physically realistic animation of complex computer graphics models fast and practical.

This thesis proposes a new approach to creating physically realistic animation that differs radically from the conventional approach of numerically simulating the equations of motion of physics-based models. We replace physics-based models by fast *emulators* which automatically learn to produce similar motions by observing the models in action. Our emulators have a *neural network* structure, hence we dub them *NeuroAnimators*. A neural network is a highly interconnected network of multi-input nonlinear processing units called "neurons" (Bishop, 1995). Neural networks provide a general mechanism for approximating complex maps in high dimensional spaces, a property that we exploit in our work.

The NeuroAnimator structure furthermore enables a new solution to the control problem associated with physics-based models. Since the neural network approximation to the physical model is differentiable, it can be used to discover the causal effects that the control forces have on the actions of the model. This knowledge is essential for the design of an efficient control learning algorithm, but it is unfortunately not available through the numerical simulation alone. For that reason the solution to the control problem through the physical simulation alone is difficult to achieve.

1.1 Overview of the NeuroAnimator Approach

Our approach is motivated by the following considerations: Whether we are dealing with rigid (Hahn, 1988; Baraff, 1989), articulated (Hodgins et al., 1995; van de Panne and Fiume, 1993), or nonrigid (Terzopoulos et al., 1987; Miller, 1988) dynamic animation models, the numerical simulation of the associated equations of motion leads to the computation of a discrete-time dynamical system of the form

$$\mathbf{s}_{t+\delta t} = \Phi[\mathbf{s}_t, \mathbf{u}_t, \mathbf{f}_t]. \qquad (1.1)$$

These (generally nonlinear) equations express the vector $\mathbf{s}_{t+\delta t}$ of state variables of the system (values of the system's degrees of freedom and their velocities) at a time δt in the future as a function Φ of the state vector $\mathbf{s}_t$, the vector $\mathbf{u}_t$ of control inputs, and the vector $\mathbf{f}_t$ of external forces acting on the system at time t.

Physics-based animation through the numerical simulation of a dynamical system governed by (1.1) requires the evaluation of the map Φ at every timestep, which usually involves a non-trivial computation. Evaluating Φ using explicit time integration methods incurs a computational cost of $O(N)$ operations, where N is proportional to the dimensionality of the state space. Unfortunately, for many dynamic models of interest, explicit methods are plagued by instability, necessitating numerous tiny timesteps δt per unit simulation time. Alternatively, implicit time-integration methods usually permit larger timesteps, but they compute Φ by solving a system of N algebraic equations, generally incurring a cost of $O(N^3)$ operations per timestep.

We propose an intriguing question: Is it possible to replace the conventional numerical simulator, which must repeatedly compute Φ, by a significantly cheaper alternative? A crucial realization is that the substitute, or emulator, need not compute the map Φ exactly, but merely approximate it to a degree of precision that preserves the perceived faithfulness of the resulting animation to the simulated dynamics of the physical model.

Neural networks (Bishop, 1995; Hertz, Krogh and Palmer, 1991) offer a general mechanism for approximating complex maps in higher dimensional spaces.[1] Our premise is that, to a sufficient degree of accuracy and at significant computational savings, trained neural networks can approximate maps Φ not just for simple dynamical systems, but also for those associated with dynamic models that are among the most complex reported in the literature to date.

The NeuroAnimator, which uses neural networks to emulate physics-based animation, learns an approximation to the dynamic model by observing instances of state transitions, as well as control inputs and/or external forces that cause these transitions. Training a NeuroAnimator is quite unlike recording motion capture data, since the network observes isolated examples of state transitions rather than complete motion trajectories. By generalizing from the sparse examples presented to it, a trained NeuroAnimator can emulate an infinite variety of continuous animations that it has never actually seen. Each emulation step costs only $O(N^2)$ operations, but it is possible to gain additional efficiency relative to a numerical simulator by training neural networks to approximate a lengthy chain of evaluations of (1.1). Thus, the emulator network can perform "super timesteps" $\Delta t = n\delta t$, typically one or two orders of magnitude larger than δt for the competing implicit time-integration scheme, thereby achieving outstanding efficiency without serious loss of accuracy.

The NeuroAnimator offers an additional bonus which has crucial consequences for animation control: Unlike the map Φ in the original dynamical system (1.1), its neural network approximation is analytically differentiable. Since the control problem is often defined as an optimization task that adjusts the control parameters so as to maximize an objective function, the differentiable properties of the neural network emulator enable us to compute the gradient of the objective function with respect to the control parameters. In fact, the derivative of the objective function with respect to control inputs is efficiently computable by applying the chain rule. Easy differentiability enables us to arrive at a remarkably fast gradient ascent optimization algorithm to compute near-optimal controllers. These controllers produce a series of control inputs $\mathbf{u}_t$ to synthesize motions satisfying prescribed constraints on the desired animation. NeuroAnimator controllers are equally applicable to controlling the original physics-based models.

1.2 Contributions

This dissertation offers a unified solution to the problems of efficiently synthesizing and controlling realistic animation using physics-based graphics models. Our contribution in this thesis is twofold:

1. We introduce and successfully demonstrate the concept of replacing physics-based models with neural network emulators. By observing the physics-based models in action, the emulators capture the visually salient characteristics of their motions, which they can then synthesize at a small fraction of the cost of numerically simulating the original models.

2. We develop a control learning algorithm that exploits the structure of the neural network emulator to quickly learn controllers through a gradient descent optimization

[1]Note that Φ in (1.1) is in general a high-dimensional map from $\Re^{s+u+f} \mapsto \Re^{s}$, where s, u, and f denote the dimensions of the state, control, and external force vectors.

process. These controllers enable either the emulators or the original physics-based models to produce motions consistent with prescribed animation goals at dramatically lower costs than competing control synthesis algorithms.

3. To minimize the approximation error of the emulation, we construct a structured NeuroAnimator that has the intrinsic knowledge about the underlying model build into it. The main subnetwork of the structured NeuroAnimator predicts the state changes of the model in its local coordinate system. A cascade of smaller subnetworks forms an interface between the main subnetwork and the inputs and outputs of the NeuroAnimator. To limit the approximation error that accumulates during the emulation of the deformable models, we introduce the regularization step that minimizes the deformation energy.

4. For complex dynamical systems with large state spaces we introduced hierarchical emulators that can be trained much more efficiently. The networks in the bottom level of this two level hierarchy are responsible for the emulation of different subsystems of the model, while the network at the top level of the hierarchy combines the results obtained by the lower level.

We have developed software for specifying and training emulators which makes use of the *Xerion* neural network simulator developed by van Camp, Plate and Hinton at the University of Toronto. Using our software, we demonstrate that the neural network emulators can be trained for a variety of some of state-of-the-art physics based models in computer graphics. Fig. 1.1 shows the models used in our research. The emulators obtain speed-ups of often as much as two orders of magnitude when compared against their physical counterparts, and yet they can emulate the models with little visible degradation to the quality of motion. We also present the solutions to a variety of non-trivial control problems synthesized through an efficient connectionist control learning algorithm that uses gradient information to drastically accelerate convergence.

1.3 Thesis Outline

Chapter 2 presents prior work in computer graphics and neural networks relevant to our research. It starts the survey with the work done in computer graphics on physics-based modeling, and proceeds to overview the neural network research upon which we draw.

Chapter 3 defines a common type of artificial neural network and discusses techniques for training it, most importantly the backpropagation algorithm—a fundamental optimization method used for neural network training as well as controller synthesis. The chapter includes extensions of this technique, and describes other optimization scenarios in relation to our work.

Chapter 4 explains the practical application of neural network concepts to the construction and training of different classes of NeuroAnimators. It introduces a strategy for building networks that avoid serious overfitting, yet are flexible enough to approximate highly nonlinear mappings. It also includes the discussion on the network input/output structure, the use of hierarchical networks to tackle physics-based models with large state spaces, and methods for changing the network structure to minimize the approximation error. The chapter additionally describes a strategy for generating independent training examples that ensure good generalization properties of the network, and lists the different optimization techniques used for neural network training.

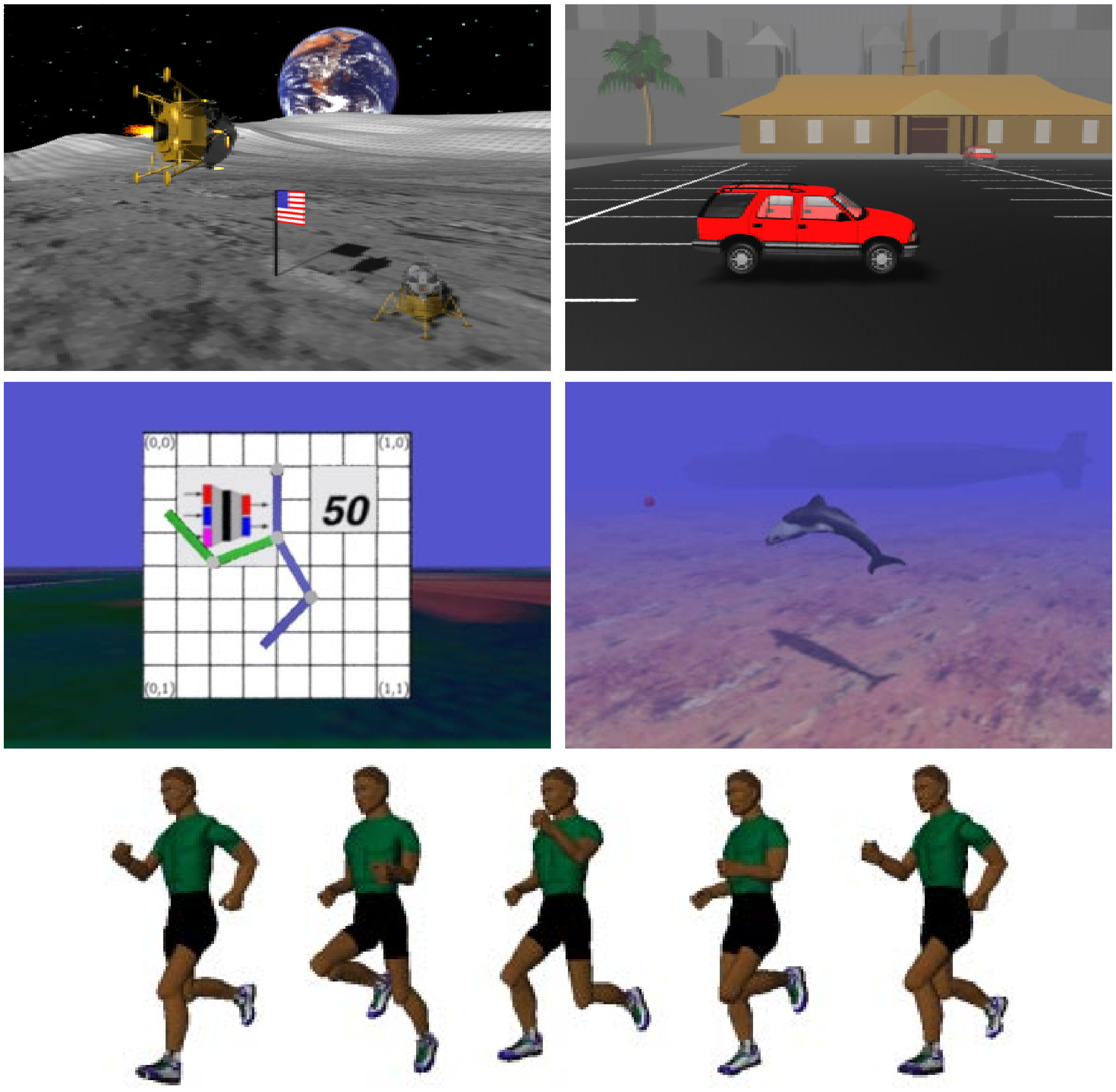


Figure 1.1: A diverse set of physical models used in our research. The collection includes some of the most complex models utilized in computer graphics research to date. The top left image shows the model of a lunar lander that was implemented as a rigid body acted upon by four independent thruster jets. The top right image shows the model of a sport utility vehicle implemented as a rigid body with constraint friction forces acting on it due to the contact with the ground. The middle left image shows the model of a multi-link pendulum under the influence of gravity. The pendulum has an independent actuator at each of its 3 joints. The middle right image shows the deformable model of a dolphin that can locomote by generating water friction forces along its body using six independent actuators. Finally, the bottom of the figure shows the model of a runner developed by Hodgins (Hodgins et al., 1995) that we used to synthesize a running motion.

In Chapter 5 we turn to the problem of control; i.e., producing physically realistic animation that satisfies goals specified by the animator. We first describe the objective function and its discrete approximation and then propose an efficient gradient based optimization procedure that computes derivatives of the objective function with respect to the control inputs through the back-propagation algorithm.

Chapter 6 presents a list of trained NeuroAnimators and supplies performance benchmarks and an error analysis for them. Additionally, the chapter discusses the use of the regularization step during the emulation of nonrigid models and its impact on the approximation error. The second part of the chapter describes the results of applying the new control learning algorithm to the trained NeuroAnimators. The report includes the comparison of the new technique with the control learning techniques used previously in computer graphics literature.

Chapter 7 concludes the thesis and presents future work.

Appendix A implements a C++ functions for calculating the outputs of a neural network from the inputs.

Appendix B derives the on-line weight update rule for a simple feedforward neural network with one hidden layer and includes a C++ implementation of the algorithm.

Appendix C derives the on-line input update rule for a simple feedforward neural network with one hidden layer and includes a C++ implementation of the algorithm.

Appendix D reviews the quaternion representation of rotation, and describes the quaternion interpolation. For the purposes of the control learning algorithm, the appendix also derives the differentiation of the quaternion rotation and defines the error metric for rotations defined as quaternions.

Appendix E describes the physical models used as the emulator prototypes. For the rigid bodies we include the SD/FAST script used to build the model and the force computation function used by the physical simulator.

Appendix F includes an example script that specifies and trains a NeuroAnimator, and it also specifies the format of the training data.

Appendix G describes the controller representation used in this thesis.

Chapter 2

Related Work

In this chapter, we present prior work in the fields of computer graphics and neural networks relevant to our research. We survey work done in computer graphics on physics-based modeling and proceed to overview the neural network research upon which we draw.

2.1 Animation Through Motion Capture

Motion capture offers an approach to physically realistic character animation that bypasses many difficulties associated with motion synthesis through numerical simulation. This method captures motion data on a computer using sensors attached to different body parts of an animal. As the animal moves, each sensor outputs to the computer a motion track that records changes over time in the sensor location. Once captured, the motion tracks can be applied to a computer model of the animal producing physically realistic motion without numerical simulation. This process can be performed very efficiently and produces highly realistic animations. The main limitation of motion capture is its static character. The motion data captured this way is hard to alter and cannot be concatenated easily. Computer animation research seeks to address these drawbacks (Bruderlin and Williams, 1995; Unuma, Anjyo and Takeuchi, 1995; Witkin and Popović, 1995; Guenter et al., 1996).

Related to our work is the idea of *synthetic motion capture* that collects motion data from synthetic models into a repertoire of kinematic actions that can be played back at interactive speeds (Lamouret and van de Panne, 1996; Yu, 1998). This approach is particularly useful when the original motion is costly to compute, as is the case for complex physics-based models. Lamouret additionally discusses the possibility of creating new animations from a set of representative example motions obtained using synthetic motion capture.

Our approach also uses synthetic models as the source of data, however, it is fundamentally different. We do not build up a database of kinematic motion sequences to be used for playback. Instead, we train a neural network emulator using examples of the state transitions to behave exactly like the physical model. Once trained, the neural network can precisely emulate the forward dynamics of the synthetic model.

2.2 Physics-Based Modeling

We seek to develop efficient techniques for producing physically realistic animations through the emulation of physics-based models. Physics-based animation involves constructing models of objects and computing their motion via physical simulation. A high degree of realism results because object motion is governed by physical laws. Additionally, since the models react to the environment in a physically plausible way, the animator does not need to spend time specifying the tedious, low-level details of the motion.

2.2.1 Modeling Inanimate Objects

Physics-based techniques have been used most successfully in the animation of inanimate objects. In particular, physics-based techniques have been applied to the animation of rigid bodies (Hahn, 1988; Baraff, 1989), articulated figures (Hodgins et al., 1995; Wilhelms, 1987), and deformable models (Terzopoulos et al., 1987; Terzopoulos and Fleischer, 1988; Desbrun and Gascuel, 1995; Carignan et al., 1992). Physics-based animation has also been applied to the simulation some more specific domains such as fluids (Kass and Miller, 1990), gases (Stam and Fiume, 1993; Foster and Metaxas, 1997), chains (Barzel and Barr, 1988), and tree leaves (Wejchert and Haumann, 1991).

A defining factor of physics-based animation is the simulation through numerical integration of equations of motion. This paradigm requires a specific model description that includes the physical properties such as mass, damping, elasticity, etc. Once the position and velocity of the model is initialized, the motion specification is completely automatic and depends entirely on the external forces acting on the model.

2.2.2 Modeling Animate Objects

There has been a substantial amount of research devoted to physics-based modeling of animate objects, such as humans and animals (Armstrong and Green, 1985; Wilhelms, 1987; Hodgins et al., 1995; Miller, 1988; Tu and Terzopoulos, 1994; Lee, Terzopoulos and Waters, 1995). The distinguishing feature that differentiates inanimate models from animate models is the ability of the latter to generate internal control forces through the use of internal actuators.

Physical models of animals are among the most complex in computer graphics. People are very sensitive to the perceptual inaccuracies in the simulation of animals, especially of humans. Therefore the complexity of the models needs to be high in order to achieve the desired realism. The complexity of a physical model leads to numerous difficulties, one of which is the time devoted to the physical simulation. The control of complex models with multiple actuators is a daunting task.

2.3 Control of Physics-Based Models

The issue of control is central to physics-based animation research. The existing methods can be divided into two groups: the *constraint based approach* and *motion synthesis.*

2.3.1 Constraint-Based Control

The constraint-based approach to control is characterized by the imposition of kinematic constraints on the motion of objects (Platt and Barr, 1988). An example of this approach is when a user specifies that a body needs to follow a certain path. There are two dominant techniques for satisfying the constraints: *inverse dynamics* and *constraint optimization.*

Inverse dynamics techniques compute a set of “constraint forces” that satisfy a set of kinematic constraints imposed by the user. This approach has been applied to control both rigid models (Isaacs and Cohen, 1987; Barzel and Barr, 1988) and deformable models (Witkin and Welch, 1990). The resulting motions are physically correct because the bodies exhibit a realistic response to forces. However, this approach is computationally very expensive.

Constraint optimization methods formulate the control problem in terms of an objective function which must be maximized over a time interval, subject to the differential equations of motion of the physical model (Brotman and Netravali, 1988; Witkin and Kass, 1988; Cohen, 1992; Liu, Gortler and Cohen, 1994). The objective function often includes a term that is inversely proportional to the control energy expenditure due to locomotion. The underlying assumption is that motions that require less energy are preferable. This method results in an open-loop controller that satisfies the constraints and maximizes the objective. This approach requires expensive numerical techniques. The need to symbolically differentiate the equations of motion renders it impractical for all but the simplest physical models.

2.3.2 Motion Synthesis

The motion synthesis approach toward locomotion control is better suited for complex physical models and appears more consistent with theories of learning in animals. Since this approach uses actuators to drive the dynamical model, it automatically constrains the control forces to the proper range and does not violate physics. This paradigm allows sensors to be freely incorporated into the models which establishes sensorimotor coupling or closed-loop control. The models can therefore react to changes in the environment and synthesize realistic controllers. To produce realistic results, motion synthesis requires high fidelity models.

Motion synthesis limits significantly the amount of control that the animator has over the model, since it adds yet another level of indirection between the control parameters and the resulting motion. Therefore, the derivation of suitable actuator control sequences becomes a fundamental task in motion synthesis. Motion specification complicates as the number of actuators grows, since getting the model to move often involves finding the right coordination between different actuators. For the most part suitable controllers have been synthesized by hand but recently a significant amount of research has been done to try to automate control synthesis.

Hand-Crafted Controllers

Manual construction of controllers involves hand crafting control functions for a set of muscles. Although generally quite difficult, this method works well on models based on animals with well known muscle activations. Miller (Miller, 1988), for example, reproduced familiar motion patterns of snakes and worms using sinusoidal contraction of successive pairs of muscles along the body of the animal. Terzopoulos *et al.* (Terzopoulos and Waters, 1990; Lee, Terzopoulos and Waters, 1995) used hand-crafted controllers to coordinate the actions of different groups of facial muscles to produce meaningful expressions. But the most impressive set of controllers developed manually for deformable models to date was constructed by Tu (Tu and Terzopoulos, 1994) for her fish model. She derived a highly realistic set of controllers that uses hydrodynamic forces to achieve forward locomotion over a range of speeds, to execute turns, and to alter body roll, pitch and yaw so that the fish can move freely within its 3D virtual world.

Hand-crafted controllers have been most often designed for rigid, articulated figures. Wilhelms (Wilhelms, 1987) developed "Virya" – one of the earliest human figure animation systems that incorporates both forward and inverse dynamics simulation. Raibert (Raibert and Hodgins, 1991) synthesized useful controllers for hoppers, kangaroos, bipeds, and quadrupeds by decomposing the problem into a set of simple manageable control tasks. Hodgins *et al.* (Hodgins et al., 1995) used similar techniques to animate a variety of motions associated with human athletics. McKenna *et al.* (McKenna and Zeltzer, 1990) used coupled oscillators to simulate different gaits of a cockroach. Brooks (Brooks, 1991) hand crafted similar controllers for his robots. Stewart and Cremer (Stewart and Cremer, 1992) created a dynamic simulation of a biped walking by defining a finite-state machine that adds and removes constraint equations. A good survey of the work reviewed here is the book 'Making Them Move' (Badler, Barsky and Zeltzer, 1991).

Manual construction of controllers is both tedious and difficult, but one can use optimization techniques to derive control functions automatically.

Controller Synthesis

The approach is inspired by the "direct dynamics" technique which was described in the control literature by Goh and Teo (Goh and Teo, 1988) and earlier references cited therein. Direct dynamics prescribes a generate-and-test strategy that optimizes a control objective function through repeated forward dynamic simulation and motion evaluation. This approach resembles trial-and-error learning process in humans and animals and is therefore often referred to as "learning". It differs from the constraint optimization approach in that it does not treat physics as constraints and it represents motion in actuator-time space and not state-time space.

The direct dynamics technique was developed further to control articulated musculoskeletal models in (Pandy, Anderson and Hull, 1992) and it has seen application in the mainstream graphics literature to the control of planar articulated figures (van de Panne and Fiume, 1993; Ngo and Marks, 1993). Pandy *et al.* (Pandy, Anderson and Hull, 1992) search the model actuator space for optimal controllers, but they do not perform global optimization. Van de Panne and Fiume (van de Panne and Fiume, 1993) use simulated annealing for global optimization. Their models are equipped with simple sensors that probe the environment and use the sensory information to influence control decisions. Ngo and Marks' (Ngo and Marks, 1993) stimulus-response control algorithm presents a similar approach. They apply the genetic algorithm to find optimal controllers. The genetic algorithm is also used in the recent work of Sims (Sims, 1994). Ridsdale (Ridsdale, 1990) reports an early effort at controller synthesis for articulated figures from training examples using neural networks. Grzeszczuk and Terzopoulos (Grzeszczuk and Terzopoulos, 1995) target state-of-the-art animate models at the level of realism and complexity of the snakes and worms of Miller (Miller, 1988) and the fish of Tu and Terzopoulos (Tu and Terzopoulos, 1994). Their models first acquire a set of basic motor skills, store them in memory using compact representations, and finally reuse them to synthesize aggregate behaviors. In ((van de Panne, 1996)), van de Panne automatically synthesizes motions for physically-based quadrupeds.

2.4 Neural Networks

Research in neural networks dates as far back as the 1960s, when Widrow and co-workers proposed networks they called *adalines* (Widrow and Lehr, 1990). The name adaline is an acronym derived from ADAptive LINear Element, and it refers to a single processing unit with threshold non-linearity. At approximately the same time, Rosenblatt (Rosenblatt, 1962) studied similar single layer networks which he called *perceptrons.* He developed a learning algorithm for perceptrons and proved its convergence. This result generated much excitement and ignited hopes that neural networks can be used as a basis for artificial intelligence. Although quite successful at solving certain problems, perceptrons failed to converge to a solution on other seemingly similar tasks. Minsky and Papert (Minsky and Papert, 1969) pointed out that the convergence theorem for single-layer networks applies only to classification problems of sets that are linearly separable, and therefore are not capable of universal computation.

Although researchers had realized that the limitations of the perceptron could have been overcome by networks having more layers of units, they failed to develop a suitable weight adjustment algorithm for training such networks. In 1986, Rumelhart, Hinton and Williams (Rumelhart, Hinton and Williams, 1986) proposed an efficient technique for training multi-layer feed-forward neural networks which they called the *backpropagation algorithm.* The algorithm defines an approximation error which is a differentiable function of the weights of the network, and computes recursively the derivatives of the error with respect to the weights. The derivatives can be used to adjust the weights so as to minimize the error. Similar algorithms had been developed by a number of researchers including Bryson and Ho (Bryson and Ho, 1969), Werbos (Werbos, 1974), and Parker (Parker, 1985).

The class of networks with two layers of weights and sigmoidal hidden units has proven to be important for practical applications. It has been shown that such networks can approximate arbitrarily well any multi-dimensional functional mapping. Many papers have appeared in the literature discussing this property including (Cybenko, 1989; Hornik, Stinchcomb and White, 1989).

Minsky and Papert (Minsky and Papert, 1969) showed that any recurrent network can be represented as a feed-forward network by simply unfolding the units of the network over time. Rumelhart *et al.* (Rumelhart, Hinton and Williams, 1986) showed the correct form of the learning rule for such a network and used it to train a simple network to be a shift register and to complete sequences. The learning algorithm is a special version of the backpropagation algorithm commonly referred to as *backpropagation through time.* In this work, the emulator networks form a special class of recurrent neural networks, and backpropagation through time constitutes the backbone of the control learning algorithm.

2.5 Connectionist Techniques for Adaptive Control

Our work is closely related to research described in the mainstream neural network literature on connectionist techniques for the adaptive control of physical robots. Barto gives a concise, yet informative, introduction to connectionist learning for adaptive control in (Barto, 1990). Motor learning is usually formulated as an optimization process in which the motor task to be learned is first specified in terms of an objective function and an optimization method is then used to compute the extremum of the function.

2.5.1 Reinforcement Learning

Reinforcement learning, an approach described by Mendel and McLaren (Mendel and McLaren, 1970), addresses the problem of controlling a system. Reinforcement learning involves two

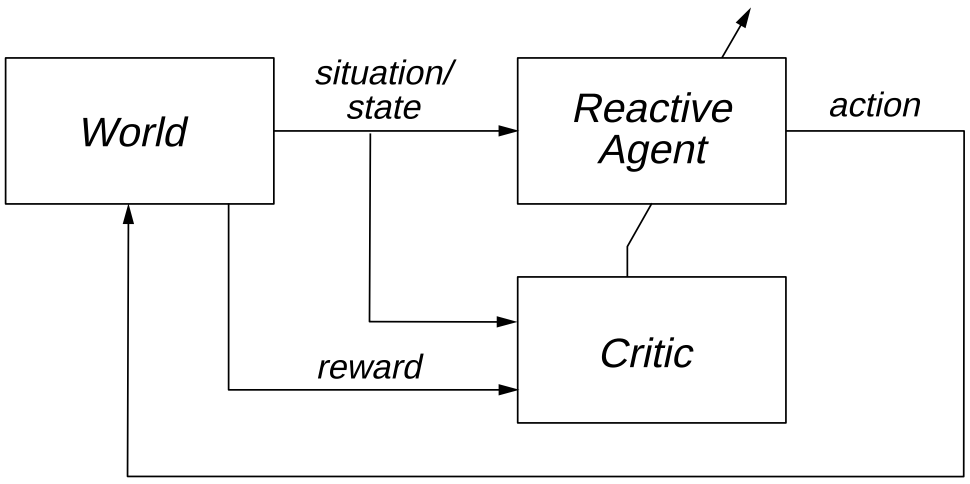


Figure 2.1: Reinforcement learning builds an approximation—the adaptive critic—that learns the effects of current actions on future events. It then uses the system performance information supplied by the critic to synthesize a controller.

problems. The first requires the construction of a *critic* that evaluates the system performance according to some control objective. The second problem is how to adjust controls based on the information supplied by the critic. Fig. 2.1 illustrates this process.

Reinforcement learning is most often used when a model of the system is unavailable and when its performance can be evaluated only by sampling the control space. This means we only have the performance signal available, but not its gradient, and we are therefore forced to search by actively exploring different control actions and incorporating those giving good results into the control rules. Reinforcement learning technique resembles learning by trial and error which selects behaviors according to its likelihood of producing reinforcement—hence the name "reinforcement learning".

Reinforcement learning builds an approximation—the adaptive critic—that learns the effects of current actions on future events (Sutton, 1984). The critic outputs the *estimate* of the total future utility which will arise from present situations and actions. Reinforcement learning performs in essence a gradient descent on the evaluation surface that it builds from the discrete examples obtained during the trial-and-error search through the control space. This approach is illustrated by the pole balancing example of Barto, Sutton and Anderson (Barto, Sutton and Anderson, 1983). Related methods that combine reinforcement learning with the gradient have also been studied (Sutton, 1984; Anderson, 1987; Barto and Jordan, 1987; Williams, 1988).

Since the active critic synthesis requires many trials, it is costly. Reinforcement is therefore inefficient, but it's very general. Werbos writes in (Werbos, 1990): "Adaptive critic is an approximation to dynamic programming which is the *only* exact and efficient method available to control motors or muscles over time so as to maximize the utility function in a noisy, nonlinear environment, without making highly specialized assumptions about this environment."

The controller synthesis techniques described in Section 2.3.2 resemble reinforcement learning in that they actively explore the control space of the system searching for optimal control actions. They differ in that they do not synthesize the active critic and are therefore less efficient.

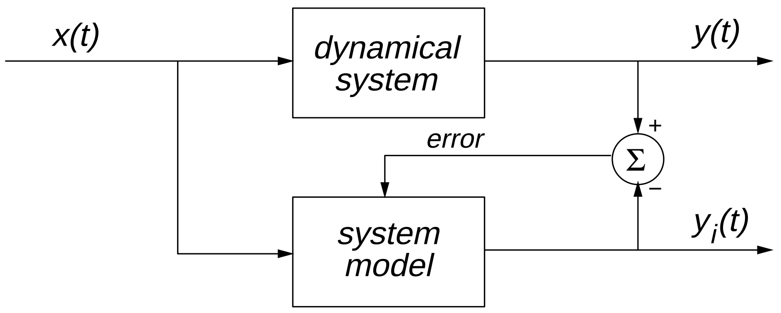


Figure 2.2: During training of the forward model, the input to the network $x(t)$ is the same as the input to the system, and the system output $y(t)$ forms the target output for the network. The neural network is trained through the use of the prediction error, $y(t) - y_i(t)$, that measures the difference between the target output of the network and its true output.

2.5.2 Control Learning Based on Connectionist Models

Connectionist control learning techniques often rely on an internal model in the form of a neural network that gives the robot system knowledge about itself and its environment. For example, the system builds an internal model of its dynamics so that it can predict its response to a force stimulation. Alternatively, it can build a model of its inverse dynamics so that it can predict a control sequence that will result in a specific action. These internal, neural network models need not be highly accurate approximations. Even an inaccurate model of inverse dynamics can provide a satisfactory control sequence whose error can be corrected with a feedback controller. Similarly, an inaccurate model of forward dynamics can be used within an internal feedback loop which corrects the controller error.

In several cases, connectionist approximations of dynamical systems have been constructed as a means to robot control. In this approach, a neural network learns to act like the dynamical system by observing the system's input-output behavior as shown in Fig. 2.2. The neural network is trained through the use of the *prediction error*: $y(t) - y_i(t)$. A neural network trained to approximate the dynamical system is referred to as the *forward model.*

A neural network can approximate the inverse of a dynamical system, which then can be used for control purposes. Jordan (Jordan, 1988) calls this the *direct inverse* approach. Fig. 2.3 shows a simple scenario where the dynamical system receives torque inputs $\mathbf{x}(t)$ and outputs the resulting trajectory $\mathbf{y}(t)$. The inverse dynamics model is set in the opposite direction, i.e., it receives $\mathbf{y}(t)$ as input and outputs $\mathbf{x}(t)$. Since the inverse model forms a map from desired outputs $\mathbf{y}(t)$ to inputs $\mathbf{x}(t)$ that produce those outputs, it is in essence a controller that can be used to control the system.

This approach was proposed and used by Albus (Albus, 1975), Widrow *et al.* (Widrow, McCool and Medoff, 1978), Miller *et al.* (Miller, Glanz and Kraft, 1987), and Jordan (Jordan, 1988). The method has been used to learn inverse kinematics (Kuperstein, 1987; Grossberg and Kuperstein, 1986), and inverse dynamics (Kawato, Setoyama and Suzuki, 1988). Atkeson and Reinkensmeyer (Atkeson and Reinkensmeyer, 1988) used content addressable memories to model the inverse dynamics, that accomplish learning in just one iteration by performing a table lookup. The first prediction application using a non-linear neural network was published in (Lapedes and Farber, 1987). See also (Weigend and Gershenfeld, 1994).

The direct inverse modeling technique has a number of limitations. Most importantly,

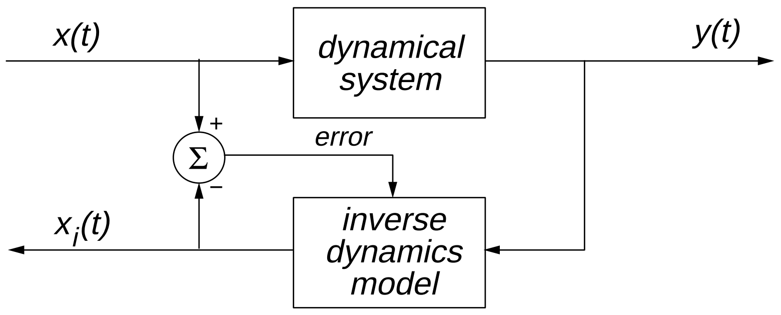


Figure 2.3: The inverse dynamics model is set in an opposite direction to that of the dynamical system.

the identification of the inverse dynamics model is problematic when the inverse is not well defined. Ill-posedness occurs when the mapping from outputs to inputs is many-to-one, in which case the network tends to average over the various targets mapping to the same input (Jordan, 1988). Additionally, the training of the inverse model requires extensive sampling of the control space in order to find an acceptable solution. However, since the control learning process requires the actual system output, extensive data collection might not always be feasible. An additional disadvantage of this approach is its off-line character.

An alternative class of algorithms learn the controller indirectly through the use of the forward model of the system. This approach comprises two stages—stage one learns the forward model as described above, stage two uses the forward model to learn the controller.

With a differentiable forward model one can solve problems of optimization over time using the *backpropagation through time* algorithm that essentially integrates the system model over time to predict future outcome and then backpropagates the error derivative back in time to compute precisely the effect of current actions on future results. When working with large sparse systems this methods permits the computation of derivatives of the utility very quickly. This technique was described by Werbos (Werbos, 1974; Werbos, 1988), Jordan and Rumelhart (Jordan and Rumelhart, 1992), and Widrow (Widrow, 1986; Nguyen and Widrow, 1989). Narendra and Parthasarathy (Narendra and Parthasarathy, 1991) propose a quicker optimization alternative to using backpropagation through time.

This method has advantages over the direct identification of the system inverse when the inverse is not well defined (Jordan and Rumelhart, 1992). The ill-posedness of the problem does not prevent it from converging to a unique solution since it uses gradient information to adjust the control signal in order to reduce the performance error. The system heads towards one of the solutions; the direct inverse, on the other hand, does not converge to a correct solution. An additional feature of this approach is the use of a uniform parameter adjustment mechanism (the backpropagation algorithm) during the system identification stage and the control learning stage.

Control learning techniques that utilize the forward model become particularly useful when the system itself is not analyzable or it is not possible to differentiate it, which is often the case for complex systems.

Forward and Inverse Modeling

Jordan and Rumelhart (Jordan and Rumelhart, 1992) proposed a scheme to combine the forward model and its inverse. The first stage of this process learns the forward model of the

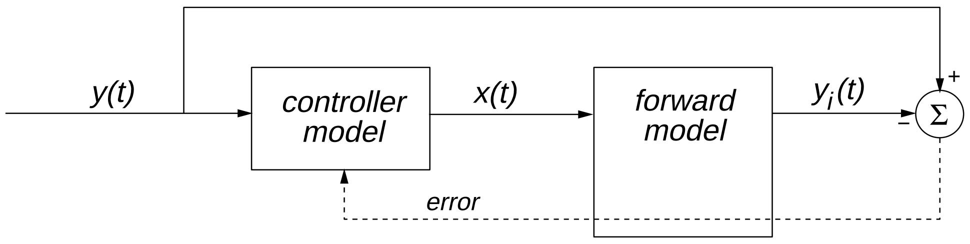


Figure 2.4: The control learning phase of the inverse modeling algorithm of Jordan and Rumelhart. This method backpropagates the prediction error through the forward model to calculate the error in the motor command, which is then used as the error signal for training the inverse. The dashed line shows the error signal pass through the forward model before reaching the inverse model during the backpropagation step.

controlled object using the technique described in Section 2.5.2. The second stage, shown in Fig. 2.4, trains the inverse dynamics model. During the control learning phase, the algorithm feeds the forward model network the desired trajectory $\mathbf{y}(t)$ which computes the feedforward motor command $\mathbf{x}_i(t)$. The prediction error, $\mathbf{y}(t) - \mathbf{y}_i(t)$, is then backpropagated through the forward model to calculate the error in the motor command, which is then used as the error signal for training the inverse. During the control learning phase the parameters of the forward model are fixed and only the parameters of the controller are adjusted. When the control learning stage is finished the controller together with the forward model form an identity transformation.

The Truck Backer-Upper

The control learning strategy presented in this thesis resembles most closely the approach used by Nguyen and Widrow in (Nguyen and Widrow, 1989) where they develop a two-stage control learning process. The first stage trains a neural network to be an emulator for the truck and trailer kinematics using the technique for system identification described in Section 2.5.2. The second stage trains a neural-network controller to control the emulator. Once the controller knows how to control the emulator, it is then able to control the actual truck.

Fig. 2.5 illustrates the procedure for adapting the controller. In the figure, the block labeled **T** represents the trailer truck emulator. The truck emulator takes as input the state of the model at time i and the steering signal, and outputs the state of the model at time $i + 1$. The block labeled **C** represents the neural network controller that takes as inputs the state of the model and outputs the steering signal for the truck. The first step of this process, shown in the top half of the figure, computes the motion trajectory of the truck for a given controller through the forward simulation. The second step of this process, shown in the bottom half of the figure, uses the trajectory obtained in the first step to evaluate the error in the objective function, and then computes the derivative of this error with respect to the controller weights using backpropagation through time.

The control learning process starts by setting the truck initial position, and choosing the controller weights at random. The truck backs up, until it stops. The final error in the truck position is used by the backpropagation algorithm to adapt the controller. The weights are changed by the sum of error deltas computed over all iterations using steepest descent.

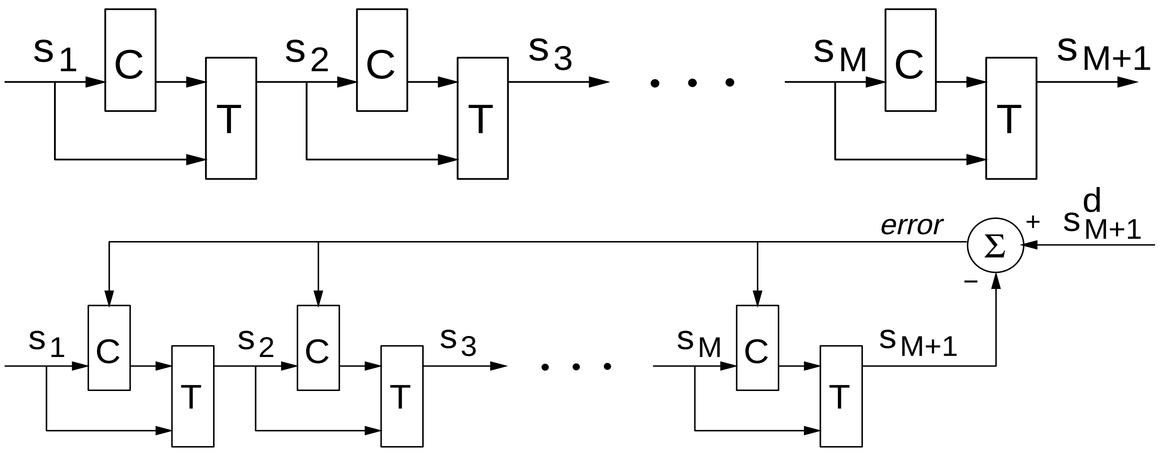


Figure 2.5: Forward emulation of the truck kinematics (top) and training of a neural-network controller to control the emulator (bottom).

The synthesized controller can park the truck in the desired position from any initial configuration. Since the solution learned by the controller is substantially general, it is rather difficult to find. To arrive at the solution, the control learning algorithm needs to process a large set of training examples and therefore converges slowly. Additionally, since the training data must have examples corresponding to different initial configurations of the truck, it must be generated off-line. Finally, the main disadvantage of the proposed controller representation is its static character—every time the objective function changes the controller needs to be completely relearned. This approach is therefore not suitable for dynamic environments where control objectives change over time.

Although related to Nguyen and Widrow's paper, the work presented in this thesis differs significantly from it: In order to achieve a better performance, the emulators have been trained to approximate a chain of evaluations of a numerical simulator as described in Section 4.1. Hierarchical emulators, discussed in Section 4.4, have been introduced to approximate highly complex, deformable models used in computer graphics.

Our technique also offers a different controller synthesis approach that works well in dynamic environments with changing control objectives. Our controllers solve the problem of getting from a given initial configuration to a given final configuration, and therefore are much easier to synthesize than the general controllers used by Nguyen and Widrow. Since our control synthesis algorithm works on-line from a few training examples that get updated after each iteration, it converges very rapidly.

2.6 Summary

To date, network architectures have found rather few applications in computer graphics. One application has been the control of animated characters. Ridsdale (Ridsdale, 1990) reports a method for skill acquisition using a connectionist model of skill memory. The sensor-actuator networks of van de Panne and Fiume (van de Panne and Fiume, 1993) are recurrent networks of units that take sensory information as input and produce actuator controls as output. Sims (Sims, 1994) employs a network architecture to structure simple "brains" that control evolved creatures. Our work differs fundamentally from these efforts; it is more closely related to the neural netowrk research on control.

As we have seen in this chapter, neural network researchers have devoted a significant amount of attention to control. Although we draw upon their work, especially that in (Nguyen and Widrow, 1989), we must adapt existing techniques to fit the requirements of computer animation. The theme of our work is to use connectionist techniques to tackle the fundamental problems of physics-based computer animation: How to produce realistic motions of complex physics-based models efficiently and how to synthesize controllers for these models.

Chapter 3

Artificial Neural Networks

In this chapter we define a common type of artificial neural network and discuss techniques for training it. We describe the backpropagation algorithm—a popular optimization method used to train neural networks and to synthesize controllers. Additionally, we present extensions of this technique, and describe other optimization scenarios in relation to our work. Finally, we outline a strategy for building networks that avoid serious overfitting, yet are flexible enough to approximate highly nonlinear mappings.

3.1 Neurons and Neural Networks

In mathematical terms, a *neuron* is an operator that maps $\Re^p \mapsto \Re$. Referring to Fig. 3.1, neuron j receives a signal z_j that is the sum of p inputs x_i scaled by associated connection weights w_{ij}:

$$z_j = w_{0j} + \sum_{i=1}^{p} x_i w_{ij} = \sum_{i=0}^{p} x_i w_{ij} = \mathbf{x}^T \mathbf{w}_j, \tag{3.1}$$

where $\mathbf{x} = [x_0, x_1, \ldots, x_p]^T$ is the input vector, $\mathbf{w}_j = [w_{0j}, w_{1j}, \ldots, w_{pj}]^T$ is the weight vector of neuron j, and w_{0j} is the bias parameter, which can be treated as an extra connection with constant unit input, $x_0 = 1$, as shown in the figure. The neuron outputs a signal $y_j = g(z_j)$, where g is a continuous, monotonic, and often nonlinear activation function, commonly the logistic sigmoid $g(z) = \sigma(z) = 1/(1 + e^{-z})$.

A *neural network* is a set of interconnected neurons. In a simple *feedforward neural network*, the neurons are organized in layers so that a neuron in layer l receives inputs only from the neurons in layer $l - 1$. The first layer is commonly referred to as the input layer and the last layer as the output layer. The intermediate layers are called hidden layers.

Fig. 3.1 shows a fully connected network with only a single hidden layer. We use this popular type of network in our algorithms. The hidden and output layers include bias units that group together the bias parameters of all the neurons in those layers. The input and output layers use linear activation functions, while the hidden layer uses the logistic sigmoid activation function. The output of the jth hidden unit is therefore given by $h_j = \sigma(\sum_{i=0}^{p} x_i v_{ij})$.

The backpropagation network used in our experiments is well suited for the approximation and the emulation of physics-based models. This type of network can estimate high-dimensional maps with a relatively small number of hidden units and therefore works efficiently. The alternative network architectures, e.g., locally-tuned networks proposed by

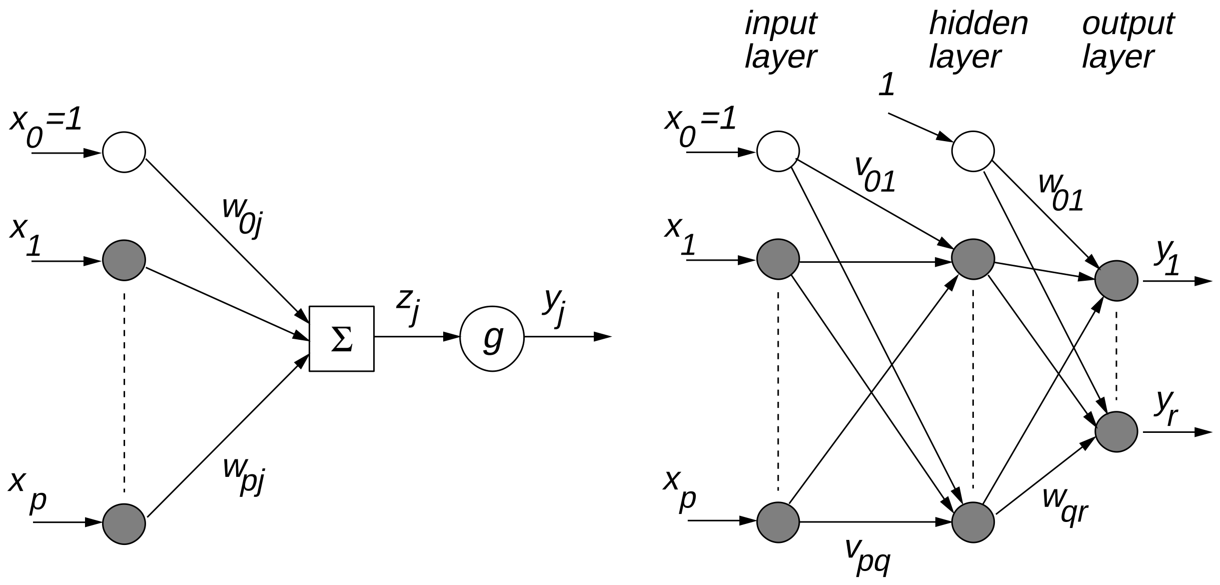


Figure 3.1: Mathematical model of a neuron j (a). Three-layer feedforward neural network $\mathbf{N}$ (b). Bias units are not shaded.

Moody and Darken (Moody and Darken, 1989), often learn very quickly but require many processing units to accurately approximate high-dimensional maps and therefore are inefficient. For our application, the slow learning rate of the backpropagation network is not a limiting factor since the emulator needs to be trained only once. However, the emulation efficiency of the backpropagation network is of primary importance during the emulation.

3.2 Approximation by Learning

We denote a 3-layer feedforward network with p input units, q hidden units, r output units, and weight vector $\mathbf{w}$ as $\mathbf{N}(\mathbf{x}, \mathbf{w})$. It defines a continuous map $\mathbf{N} : \Re^p \mapsto \Re^r$. With sufficiently large q, a feedforward neural network with this architecture can approximate as accurately as necessary any continuous map $\Phi : \Re^p \mapsto \Re^r$ over a compact domain $\mathbf{x} \in \mathcal{X}$ (Cybenko, 1989; Hornik, Stinchcomb and White, 1989); i.e., for an arbitrarily small $\epsilon > 0$ there exists a network $\mathbf{N}$ such that

$$\forall \mathbf{x} \in \mathcal{X}, \quad E(\mathbf{x}, \mathbf{w}) = \|\Phi(\mathbf{x}) - \mathbf{N}(\mathbf{x}, \mathbf{w})\|^2 < \epsilon, \tag{3.2}$$

where E is the approximation error.

A neural network can *learn* an approximation to a map Φ by observing training data consisting of input-output pairs that sample Φ. The training sequence is a set of *examples*, such that the τth example comprises the pair

$$\begin{cases} \mathbf{x}^\tau = [x_1^\tau, x_2^\tau, \ldots, x_p^\tau]^T; \\ \mathbf{y}^\tau = \Phi(\mathbf{x}^\tau) = [y_1^\tau, y_2^\tau, \ldots, y_r^\tau]^T \end{cases} \tag{3.3}$$

where $\mathbf{x}^\tau$ is the input vector and $\mathbf{y}^\tau$ is the associated desired output vector. The goal of training is to utilize the examples to find a set of weights $\mathbf{w}$ for the network $\mathbf{N}(\mathbf{x}, \mathbf{w})$

such that, for all inputs of interest, the difference between the network output and the true output is sufficiently small, as measured by the approximation error (3.2).

3.3 Backpropagation Algorithm

The backpropagation algorithm is central to much current work on learning in neural networks. Invented independently several times, by Bryson and Ho (Bryson and Ho, 1969), Werbos (Werbos, 1974), and Parker (Parker, 1985), it has been popularized by Rumelhart, Hinton and Williams (Rumelhart, Hinton and Williams, 1986). The algorithm describes an efficient method for updating the weights of a multi-layer feedforward network to learn a training set of input-output pairs $(\mathbf{x}^\tau, \mathbf{y}^\tau)$.

A crucial realization leading to the backpropagation algorithm is that the neural network output forms a continuous, differentiable function of the network inputs and weights. Based on this fact, there exists a practical, recursive method for computing the derivatives of the outputs with respect to the weights. The derivatives are then used to adjust the weights so that the network learns to produce the correct outputs for each input vector in the training set. This is called the *weight update rule.*

The traditional backpropagation algorithm can be used to compute the derivatives of the network outputs with respect to its inputs, assuming fixed weights. This gradient information tells us how to adjust the network inputs in order to produce the desired outputs, and is therefore very important for control learning where one often needs to find a set of control inputs that will yield a specific state output. This is called the *input update rule.* It forms the essential step of the control learning algorithm described in Section 5.

In the next two sections, we first outline the weight update rule, then the input update rule.

3.3.1 Weight Update Rule

The weight update rule adjusts the network weights so that the network learns an example set $(\mathbf{x}^\tau, \mathbf{y}^\tau)$. For input vector $\mathbf{x}^\tau \in \mathbf{X}$ and weight vector $\mathbf{w} \in \mathbf{W}$, the algorithm defines the *network approximation error* as

$$E^\tau(\mathbf{w}) = E(\mathbf{x}^\tau, \mathbf{w}) = ||\Phi(\mathbf{x}^\tau) - \mathbf{N}(\mathbf{x}^\tau, \mathbf{w})||^2, \tag{3.4}$$

and it seeks to minimize the objective

$$E(\mathbf{w}) = \frac{1}{2} \sum_{\tau=1}^{n} E^\tau(\mathbf{w}), \tag{3.5}$$

where n is the number of training examples. The simplest implementation of the algorithm uses gradient descent to obtain the update rule. The on-line version of the algorithm adjusts the network weights after each training example τ:

$$\mathbf{w}^{l+1} = \mathbf{w}^l - \eta_w \nabla_{\mathbf{w}} E^\tau(\mathbf{w}^l) \tag{3.6}$$

where $\eta_w < 1$ denotes the *weight update learning rate*, and l denotes the current iteration of the algorithm. Fig. 3.2 illustrates this process.

Appendix B derives the on-line weight update rule for a simple feedforward neural network with one hidden layer and includes a C++ implementation of the algorithm.

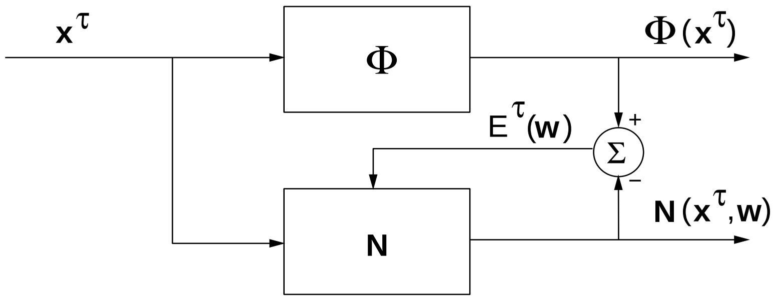


Figure 3.2: The backpropagation algorithm learns a map Φ by adjusting the weights $\mathbf{w}$ of the network $\mathbf{N}$ in order to reduce the difference between in the network output $\mathbf{N}(\mathbf{x}^\tau, \mathbf{w})$ and the desired output $\Phi(\mathbf{x}^\tau)$. Depicted here is the on-line version of the algorithm that adjusts the weights of the network after observing each training example.

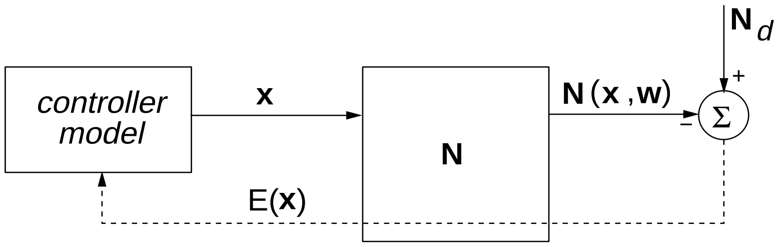


Figure 3.3: The network computes the derivative of $E(\mathbf{x})$ with respect to the input $\mathbf{x}$ by backpropagating the error signal through the network $\mathbf{N}$. The derivative is used to adjust the inputs in order to minimize the error.

3.3.2 Input Update Rule

The input update rule adjusts the network inputs to produce the desired network outputs. It assumes that the weights of the network have been trained to approximate a map Φ, and that they are fixed during the input adjustment step. During the input update rule we seek to minimize the objective defined as

$$E(\mathbf{x}) = ||\mathbf{N}_d - \mathbf{N}(\mathbf{x})||^2, \tag{3.7}$$

where $\mathbf{N}_d$ denotes the desired network output. The simplest implementation of the algorithm uses gradient descent to obtain the update rule. The on-line version of the algorithm adjusts the inputs using the following rule

$$\mathbf{x}^{l+1} = \mathbf{x}^l - \eta_x \nabla_{\mathbf{x}} E(\mathbf{x}^l) \tag{3.8}$$

where $\eta_x < 1$ denotes the *input update learning rate*, and l denotes the iteration of the algorithm. Fig. 3.3 illustrates this process.

Appendix C derives the on-line input update rule for a simple feedforward neural network with one hidden layer and includes a C++ implementation of the algorithm.

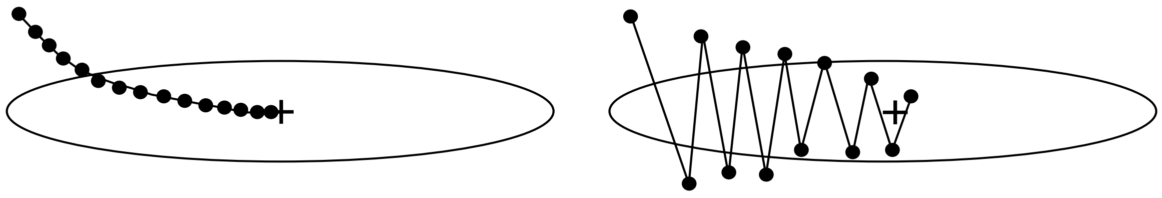


Figure 3.4: The gradient descent on a simple quadratic surface of two variables. The surface minimum is at +, and the ellipses show the contours of constant error. The left trajectory was produced using the gradient descent with a small learning rate. The solution moves towards the minimum in tiny steps. The right trajectory uses the gradient descent with a larger learning rate. The solution moves towards the solution slowly due to wide oscillations.

3.4 Alternative Optimization Techniques

The simple gradient descent used in the update rules (3.6) and (3.8) can be very slow if the learning rate is small, and it can oscillate widely if the learning rate η is too large. Fig. 3.4 illustrates this idea for a simple quadratic surface of two variables.

However, the performance of the gradient descent algorithm can be improved, or in some cases the algorithm can be replaced altogether with a more efficient optimization technique. We briefly describe here some popular optimization techniques used in the connectionist literature. For a more thorough overview, we refer the reader to many standard textbooks which cover the non-linear optimization techniques, including (Polak, 1971; Gill, Murray and Wright, 1981; Dennis and Schnabel, 1983; Luenberger, 1984; Fletcher, 1987). Also, (Hinton, 1989) offers a concise but thorough overview of the connectionist learning techniques.

This chapter explains the difference between the on-line mode and the batch mode training and presents the optimization techniques relevant to our work. Finally, it describes possible improvements to the gradient descent algorithm, and reviews some more sophisticated optimization algorithms.

All the algorithms presented in this chapter are described in terms of the weight updates. If an algorithm can be also used for the input updates, we state it without writing the equations explicitly.

3.4.1 On-Line Training vs. Batch Training

The on-line implementation of the backpropagation algorithm updates the optimization parameters after each training example, as in Equations (3.6) and (3.8). However, if the training data can be generated before the network training initiates, it often is more practical to use the batch mode. In the batch mode, the optimization parameters get updated once after all the examples have been presented to the network. The corresponding batch mode weight update rule can be expressed mathematically as

$$\mathbf{w}^{l+1} = \mathbf{w}^l - \eta_w \nabla_{\mathbf{w}} E(\mathbf{w}^l), \tag{3.9}$$

where $E(\mathbf{w})$ was defined in (3.5).

Generally, it is advantageous to use the off-line training method since it often converges faster than the on-line training method. Additionally, some of the techniques described below work only in the batch mode. However, on-line training is useful in certain situations.

Since the on-line training behaves like a stochastic process if the patterns are chosen in a random order, it can produce better results than the batch learning when applied to large networks with many weights that tend to have numerous local minima and saddle points. Additionally, the on-line training works faster on problems with highly redundant data. In practice, we found that a hybrid approach that divides the training data into small batches, each having about 30-40 training examples, works best on large networks. Unlike the batch mode algorithm that updates the network weights only after processing all the training examples, this algorithm makes a modification after evaluating the examples in each mini-batch. This simple strategy often exceeds the performance of a batch mode algorithm when applied to a complex network that requires a large data set, and for which the batch mode updates become exceedingly rare.

3.4.2 Momentum

The strategy of this method is to give each optimization parameter, whether it is a weight or an input, some inertia or momentum, so that it changes in the direction of the time-averaged downhill force that it feels, instead of rapidly changing the descent direction after each update. This simple strategy increases the effective learning rate without oscillating like the simple gradient descent. It augments the gradient descent update rule with a *momentum term*, which chooses the descent direction of the optimization process based on the previous trials. After each example τ, the momentum term computes the weight deltas using the following formula

$$\delta \mathbf{w}^{l+1} = -\eta_w \nabla_{\mathbf{w}} E^{\tau}(\mathbf{w}^l) + \alpha_w \delta \mathbf{w}^l, \tag{3.10}$$

where the momentum parameter α_w must be between 0 and 1. The on-line implementation of this technique updates the weights after each training example

$$\mathbf{w}^{l+1} = \mathbf{w}^l + \delta \mathbf{w}^{l+1}. \tag{3.11}$$

In the batch mode, the weights are updated after all the training examples

$$\mathbf{w}^{l+1} = \mathbf{w}^l + \sum_{\tau=1}^{n} \delta \mathbf{w}^{l+1}. \tag{3.12}$$

Interestingly, there is a close relationship between the update rule that uses the momentum term and the law of inertia. To draw this parallel, let imagine a point mass moving through the weight space under the force of gravity. Let $\mathbf{w}^l$ denote the position of the point in the weight-space at time l, and $\delta \mathbf{w}^l$ the change in position, or velocity, of the point at time l. We can now interpret (3.10) as describing the point mass acceleration generated by the gravity force represented in the equation by the gradient term. According to this interpretation, the momentum term produces a "physically correct" update rule that applies a force to change the velocity of the point. The same physical analogy applied to the standard gradient descent method (Eq. 3.6) produces a "physically incorrect" update rule that uses the "gradient force" to change the position of the point mass, instead of its velocity.

The use of the momentum term is not only physically more appealing, it also has practical advantages. The new update rule reduce oscillations in the optimization trajectory, but it also increases the learning rate if the oscillations are not present. If at every step the algorithm moves in the same direction, then the effective learning rate becomes $\eta_w/(1-\alpha_w)$.

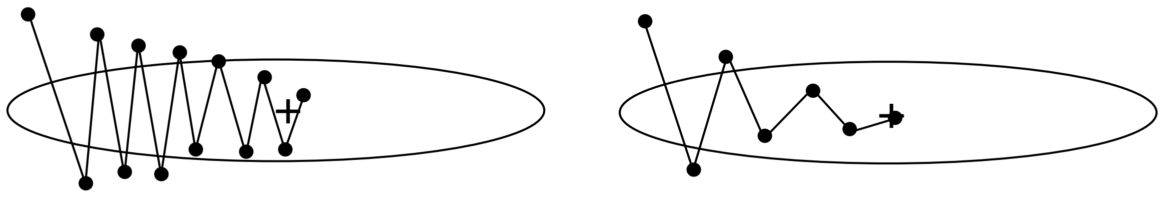


Figure 3.5: The gradient descent on a simple quadratic surface of Fig. 3.4. The left trajectory was generated using simple gradient descent and therefore oscillates widely. The right trajectory was produced by adding the momentum term. In this case the oscillations become damped and the algorithm converges much more quickly towards the minimum.

Since α_w is usually chosen to be 0.9, this leads to a ten times higher learning rate than simple gradient descent.

The momentum term is useful in both on-line and batch mode updating. We use it both for the training of networks (the mini-batch weight update), and for the control learning (input update rule used by the control learning algorithm).

3.4.3 Line Searches

Function minimization can be made much more efficient through the use of *line searches* along selected directions. At each iteration of the algorithm using the line search, we move along a fixed direction $\mathbf{d}$, but we can perform multiple function evaluations to determine the optimal stepsize λ^l

$$\mathbf{w}^{l+1} = \mathbf{w}^l + \lambda^l \mathbf{d}^l. \tag{3.13}$$

One common approach that uses a line search is the *steepest descent method.* The algorithm descends along the gradient direction $\mathbf{d}^l = -\nabla_{\mathbf{w}} E(\mathbf{w}^l)$, but it updates the stepsize λ^l through a minimization done along the line of descent. Although the algorithm requires multiple function evaluations at each iteration, it generally converges in far fewer steps than simple gradient descent.

At each iteration of steepest descent the new gradient direction is perpendicular to its predecessor, thus the algorithm approaches the minimum along a zigzag path. A variant of the steepest descent algorithm—the *conjugate gradient method*—chooses the new search direction to be a compromise between the gradient direction and the previous search direction

$$\mathbf{d}^{l+1} = -\nabla E_{\mathbf{w}}(\mathbf{w}^l) + \beta \mathbf{d}^l \tag{3.14}$$

where one possible choice for β is the *Polak-Ribiere variant*

$$\beta = \frac{(\nabla E_{\mathbf{w}}(\mathbf{w}^l) - \nabla E_{\mathbf{w}}(\mathbf{w}^{l-1}))^T \nabla E_{\mathbf{w}}(\mathbf{w}^l)}{\nabla E_{\mathbf{w}}(\mathbf{w}^{l-1})^2}. \tag{3.15}$$

When transformed by the Hessian, the conjugate gradient method, applied to an n-dimensional problem, keeps the last n search directions mutually perpendicular, and therefore each new line search spoils as little as possible the result achieved by the previous steps. On a quadratic surface in n dimensions this technique reaches the minimum in exactly n iterations. Fig. 3.6 illustrates steepest descent and conjugate gradient on a simple quadratic surface.

Conjugate gradient and gradient descent with the momentum term share an interest-

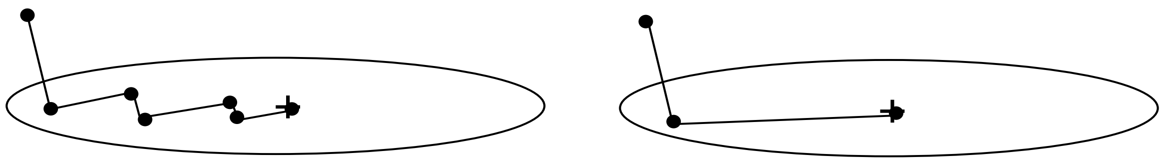


Figure 3.6: Line minimizations on a simple quadratic surface of Fig. 3.4. The left trajectory was generated using steepest descent and therefore zigzags towards the minimum. The right trajectory was produced by the conjugate gradient method and therefore reaches the minimum in exactly two steps.

ing property—neither technique precisely follows the gradient direction. Because of this relationship, gradient descent with the momentum term can be thought of as a rough approximation to conjugate gradient.

3.5 The Xerion Neural Network Simulator

Most of the techniques described here have been implemented as part of a neural network simulator called *Xerion* which was developed at the University of Toronto and is available publically.[1] Public availability of software such as Xerion contributes to making our NeuroAnimator approach easily accessible to the graphics community.

Xerion is a collection of C libraries that can be used to train many types of neural networks. The simulator makes the construction of complex networks simple through the use of a command line interface. The interface is written using **Tcl** and **Tk**—a simple scripting language developed by John Ousterhout at the University of California at Berkley that can be easily extended.

Xerion describes networks using a set of objects. The main top-level objects are `nets` that define the structure of the network. Inside `nets` are `layers` that define groups that combine the units with common characteristics. Finally, at the bottom-level of the hierarchy are `units` and `links`.

The other top-level objects are: example sets that contain the data necessary to train and test a network, and minimizers that train nets on example sets using any of several optimization techniques. Xerion interpreter has commands for building and manipulating these objects. Appendix F gives an example of a Xerion script.

[1]Available from `ftp://ftp.cs.toronto.edu/pub/xerion`

Chapter 4

From Physics-Based Models to NeuroAnimators

This chapter explains the practical application of neural network concepts to the construction and training of different classes of NeuroAnimators. This includes network input/output structure, the issue of network size in proper fitting of the training data, and the use of hierarchical networks to tackle physics-based models with large state spaces. The last section presents a sequence of modifications that we apply to the emulator to improve its approximation quality. This includes the notation used to present the operations, a detailed description of each transformation, and finally the architecture of this structured NeuroAnimator.

4.1 Emulation

Our task is to construct neural networks that approximate Φ in the dynamical system governed by (1.1). We propose to employ backpropagation to train feedforward networks $\mathbf{N}_\Phi$ to predict future states using super timesteps $\Delta t = n\delta t$ while containing the approximation error so as not to appreciably degrade the visual realism of the resulting animation. Analogous to (1.1), the basic emulation step is

$$\mathbf{s}_{t+\Delta t} = \mathbf{N}_\Phi[\mathbf{s}_t, \mathbf{u}_t, \mathbf{f}_t]. \tag{4.1}$$

The trained emulator network $\mathbf{N}_\Phi$ takes as input the state of the model, its control inputs, and the external forces acting on it at time t, and produces as output the state of the model at time $t + \Delta t$ by evaluating the network. A sequence of such evaluations constitutes the emulation process. After each evaluation, the network control and force inputs receive new values, and the network state inputs receive the emulator outputs from the previous evaluation. Fig. 4.1 illustrates the emulation process. The figure represents each emulation step by a separate network whose outputs become the inputs to the next network. In reality, the emulation process uses a single network that uses its outputs as inputs to the subsequent evaluation step.

Since the emulation step is large compared with the physical simulation step, we often find the sampling rate of the motion trajectory produced by the emulator to be too coarse for the purposes of animation. To deal with the problem, we sample the motion trajectory at the animation frame rate computing the desired states through linear interpolation of

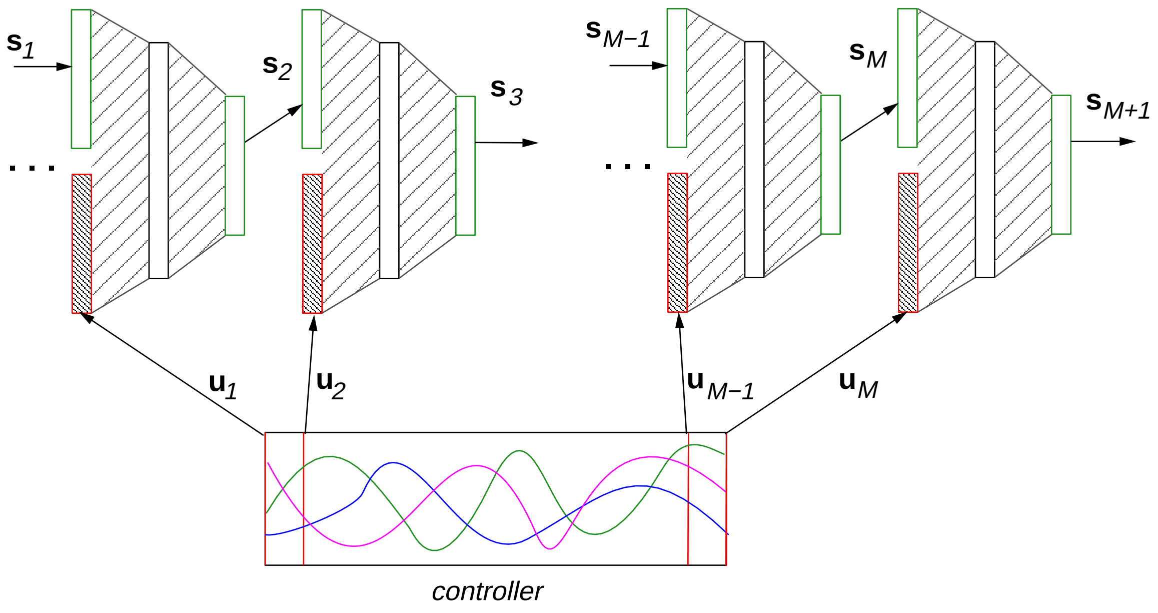


Figure 4.1: Forward dynamics emulation using a neural networks. At each iteration, the NeuroAnimator output becomes the state input at the next iteration of the algorithm. Note that the same network is used iteratively.

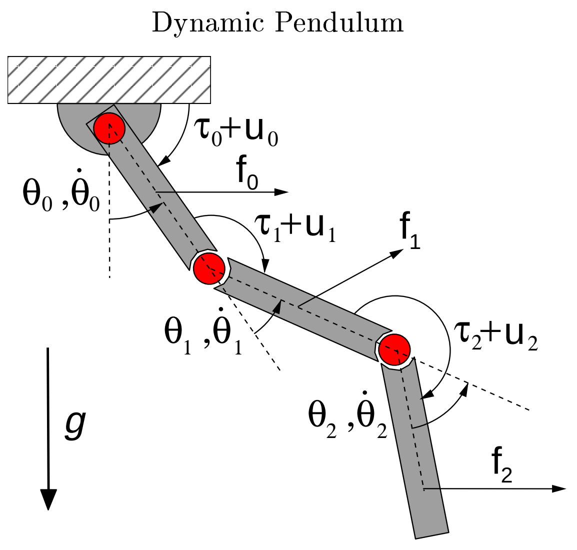
samples obtained from the emulation. Linear interpolation produces satisfactory smooth motion, although a more sophisticated scheme could improve the result.

4.2 Network Structure

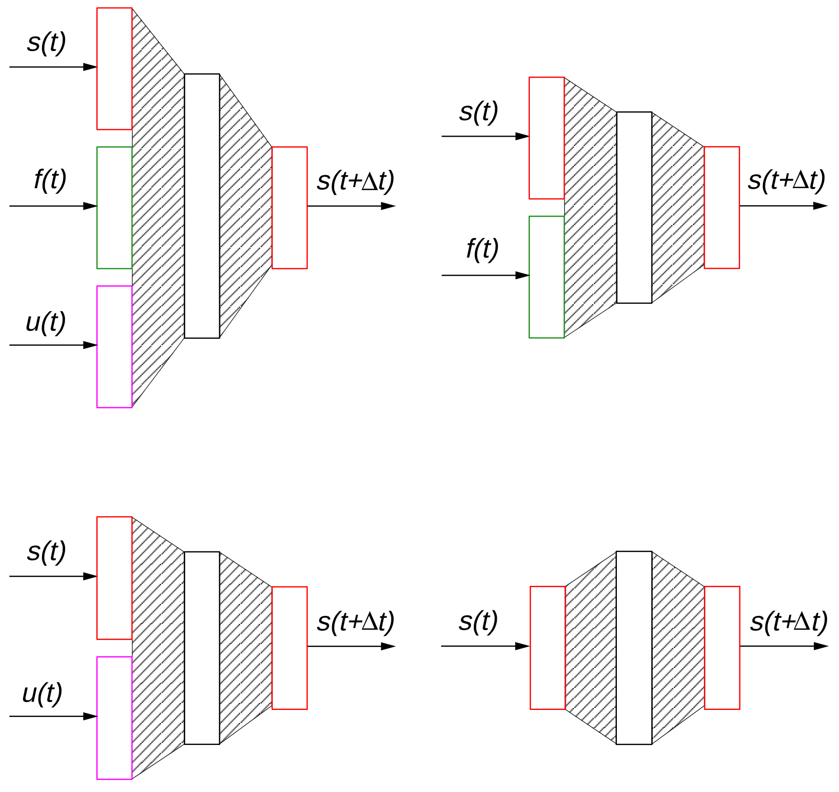
The emulator network has a single set of output variables specifying $\mathbf{s}_{t+\Delta t}$. The number of input variable sets depends on whether the physical model is active or passive and the type of forces involved. A dynamical system of the form (1.1), such as the multi-link pendulum illustrated in Fig. 4.2(a), with control inputs $\mathbf{u}$ comprising joint motor torques is known as active, otherwise, it is passive. If we wish, in the fully general case, to emulate an active model under the influence of unpredictable applied forces, we employ a full network with three sets of input variables: $\mathbf{s}_t$, $\mathbf{u}_t$, and $\mathbf{f}_t$, as shown in the figure. For passive models, the control $\mathbf{u}_t = \mathbf{0}$ and the network simplifies to one with two sets of inputs, $\mathbf{s}_t$ and $\mathbf{f}_t$.

In the special case when the forces $\mathbf{f}_t$ are completely determined by the state of the system $\mathbf{s}_t$, we can suppress the $\mathbf{f}_t$ inputs allowing the network to learn the effects of these forces from the state transition training data. For example, the active multi-link pendulum illustrated in Fig. 4.2(a) is under influence of gravity $\mathbf{g}$ and joint friction forces $\boldsymbol{\tau}$. However, since both $\mathbf{g}$ and $\boldsymbol{\tau}$ are completely determined by $\mathbf{s}_t$, they do not have to be given as the emulator inputs. A simple emulator with two input sets $\mathbf{s}_t$ and $\mathbf{u}_t$ can learn the response of the multi-link pendulum to those external forces.

The simplest type of emulator has only a single set of inputs $\mathbf{s}_t$. This emulator can approximate passive models acted upon by deterministic external force. Fig. 4.2(b) illustrates different emulator input/output structures.


Dynamic Pendulum
τ_0+u_0
f_0
$\theta_0, \dot{\theta}_0$
τ_1+u_1
f_1
$\theta_1, \dot{\theta}_1$
τ_2+u_2
$\theta_2, \dot{\theta}_2$
f_2
g

(a)



(b)

Figure 4.2: Three-link physical pendulum and network emulators. (a) An active pendulum with joint friction τ_i, motor torques u_i, and applied forces $\mathbf{f}_i$ and gravity $\mathbf{g}$. Without motor torques, the pendulum is passive. (b) Different types of emulators.

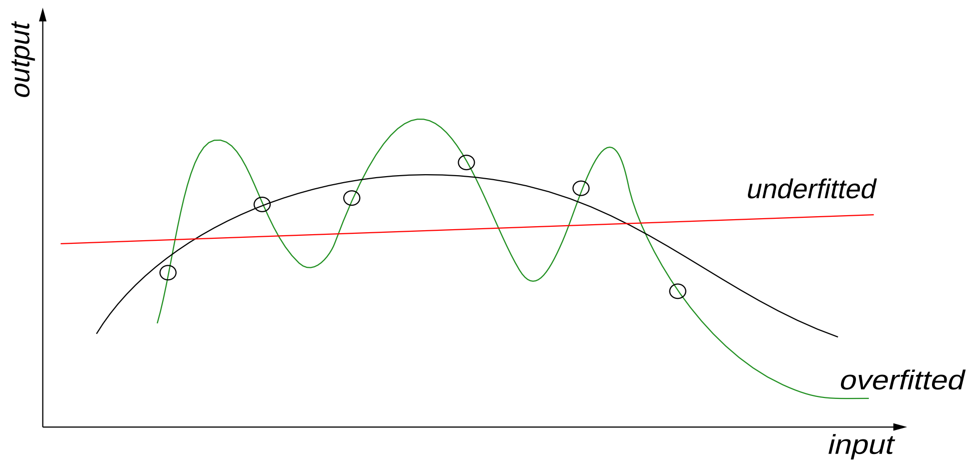


Figure 4.3: Depicted in a low-dimensional setting, a neural network with too few neurons underfits the training data. One with too many neurons overfits the data. The solid curve represents a properly chosen network which provides a good compromise between approximation (fits the training data) and generalization (generates reasonable output values away from the training examples).

4.2.1 Weights, Hidden Units, and Training Data

This section describes at a general level some of the issues relating to the emulator design. We explain how we choose the network size and its structure, how we determine the number of hidden units, and how many training examples we use to train a network. Subsequent chapters elaborate on some of the issues raised here.

Training of a neural network to approximate a functional mapping is analogous to fitting a polynomial to data and it suffers from the same problems. Mainly, a network with two few free parameters (weights) will underfit the data, while a network with too many free parameters will overfit the data. Duda and Hart (Duda and Hart, 1973) described this problem thus: "In general, reliable interpolation or extrapolation cannot be obtained unless the solution is overdetermined." Fig. 4.3 depicts these problems in a low-dimensional scenario. In our case, the underfitting is not an issue because we use networks with a sufficient number of weights. To avoid overfitting, we make sure that we use sufficient training data. We use 8-10 times as many examples as there are weights in the network, which seems sufficient to avoid serious overfitting or underfitting.

The optimal number of the hidden units depends on the dimensionality of the input and output, the amount of data available, and the predictability of the output from the input. A network with more hidden units is more flexible, and it therefore can approximate a mapping more accurately. Unfortunately, the additional units also make the network training harder, because a larger network prolongs the evaluation time, requires a larger training set, and needs more iterations of the learning algorithm for the convergence.

The choice of the number of hidden units is therefore a tradeoff between the necessary flexibility of the model and the time it takes to train the network. We usually start with a relatively small number of hidden units and increase it until we are satisfied with the approximation quality of the network. Unfortunately, the network needs to be fully retrained after each modification of its structure.

Networks that have one hidden layer of sigmoid units can approximate any functional mapping to an arbitrary accuracy. However, the networks with multiple layers of hidden units can model some more complex input-output mappings with a smaller number of weights (LeCun et al., 1989). Unfortunately, training of such networks is hard and one often has to spend an extra time designing the structure of the network to reflect the underlying problem. The emulation problem that we are trying to solve does not have a well defined structure—the number of inputs and outputs, as well as their values, vary a lot between the different instances of the problem. Despite the difficulties we build a structured network that does reflect some of the emulation invariances. Section 4.3 describes the detailed structure of a special emulator that minimizes the approximation error.

4.3 Input and Output Transformations

The accurate approximation of complex functional mappings using neural networks can be challenging. We have observed that a simple feedforward neural network with a single layer of sigmoid units has difficulty producing an accurate approximation to the dynamics of physical models. In practice, we often must transform the emulator to ensure a good approximation of the map Φ, as we explain next.

A fundamental problem is that the state variables of a dynamical system can have a large dynamic range (e.g., the position and velocity of an unconstrained particle can take values from $-\infty$ to $+\infty$). A single sigmoid unit is nonlinear only over a small region of its input space and approximately constant elsewhere. To approximate a nonlinear map Φ accurately over a large domain, we would need to use a neural network with many sigmoid units, each shifted and scaled so that their nonlinear segments cover different parts of the domain. The direct approximation of Φ is therefore impractical.

A successful strategy is to train networks to emulate *changes* in state variables rather than their actual values, since state changes over small timesteps will have a significantly smaller dynamic range. Hence, in Fig. 4.4(a) we restructure our simple network $\mathbf{N}_\Phi$ as a network $\mathbf{N}_\Phi^\Delta$ which is trained to emulate the change in the state vector $\Delta\mathbf{s}_t$ for given state, external force, and control inputs, followed by an operator $\mathbf{T}_y^\Delta$ that computes $\mathbf{s}_{t+\Delta t} = \mathbf{s}_t + \Delta\mathbf{s}_t$ to recover the next state.

We can further improve the approximation power of the emulator network by exploiting natural invariances. In particular, note that the map Φ is invariant under rotation and translation; i.e., the state changes are independent of the absolute position and orientation of the physical model relative to the world coordinate system. Hence, in Fig. 4.4(b) we replace $\mathbf{N}_\Phi^\Delta$ with an operator $\mathbf{T}'_x$ that converts the inputs from the world coordinate system to the local coordinate system of the model, a network $\mathbf{N}'_\Phi$ that is trained to emulate state changes represented in the local coordinate system, and an operator $\mathbf{T}'_y$ that converts the output of $\mathbf{N}'_\Phi$ back to world coordinates.

A final improvement in the ability of the NeuroAnimator to approximate the map Φ accrues from the normalization of groups of input and output variables. Since the values of state, force, and control variables can deviate significantly, their effect on the network outputs is uneven, causing problems when large inputs must have a small influence on outputs. To make inputs contribute more evenly to the network outputs, we normalize groups of variables so that they have zero means and unit variances. With normalization, we can furthermore expect the weights of the trained network to be of order unity and they can be given a simple random initialization prior to training. Hence, in Fig. 4.4(c) we replace $\mathbf{N}'_\Phi$ with an operator $\mathbf{T}_x^\sigma$ that normalizes its inputs, a network $\mathbf{N}_\Phi^\sigma$ that assumes

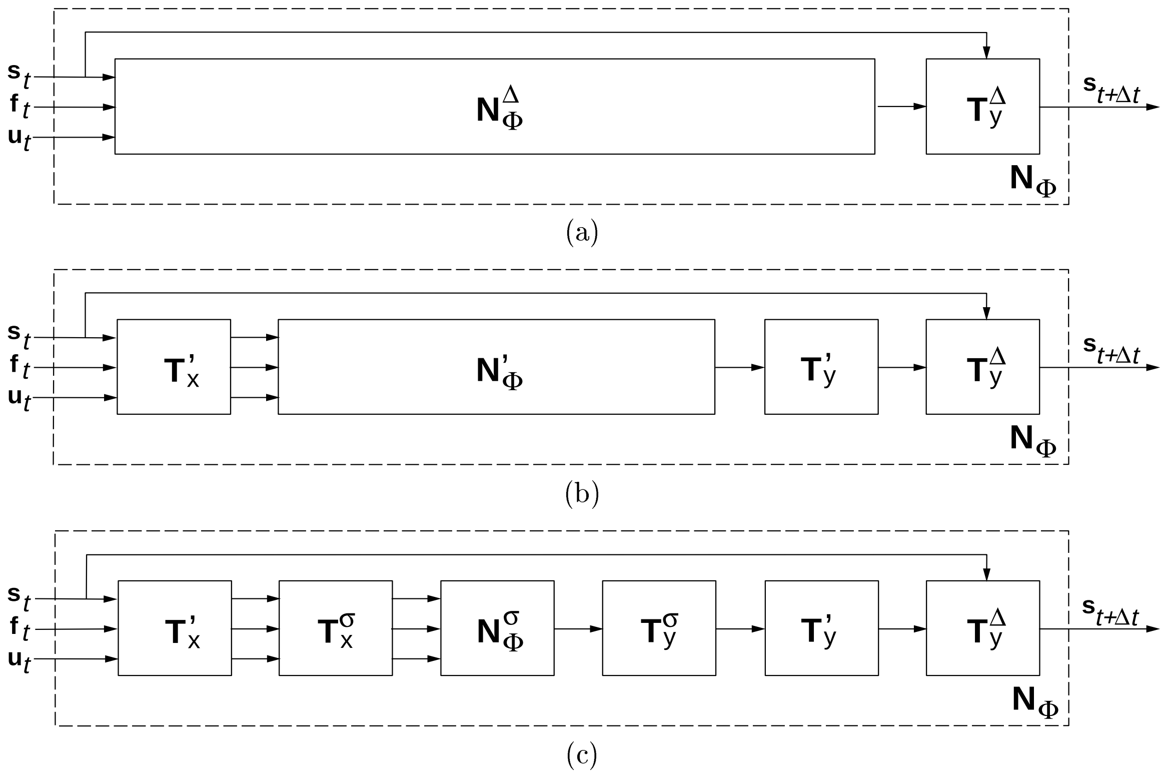


Figure 4.4: Transforming a simple feedforward neural network $\mathbf{N}_\Phi$ into a practical emulator network $\mathbf{N}_\Phi^\sigma$ that is easily trained to emulate physics-based models. The following operators perform the appropriate pre- and post-processing: $\mathbf{T}_x'$ transforms inputs to local coordinates, $\mathbf{T}_x^\sigma$ normalizes inputs, $\mathbf{T}_y^\sigma$ unnormalizes outputs, $\mathbf{T}_y'$ transforms outputs to global coordinates, $\mathbf{T}_y^\Delta$ converts from a state change to the next state (see text).

zero mean, unit variance inputs and outputs, and an operator $\mathbf{T}_y^\sigma$ that unnormalizes the outputs to recover their original distributions.

Although the final emulator in Fig. 4.4(c) is structurally more complex than the standard feedforward neural network $\mathbf{N}_\Phi$ that it replaces, the operators denoted by the letter $\mathbf{T}$ are completely determined by the state of the model and the distribution of the training data, and the emulator network $\mathbf{N}_\Phi^\sigma$ is much easier to train.

The next section introduces the notation used to describe the transformations discussed above. Following this section, each transformation is described in detail.

4.3.1 Notation

This section introduces the notation used in the ensuing derivation. Consider a simple rigid-body model. We represent the state of the model at time t as $\mathbf{s}_t = [\mathbf{p}_t, \mathbf{q}_t, \mathbf{v}_t, \boldsymbol{\omega}_t]$ where $\mathbf{p}_t$ denotes the position, $\mathbf{q}_t$ the orientation, $\mathbf{v}_t$ the velocity, and $\boldsymbol{\omega}_t$ the angular velocity of the body. We specify all orientations using quaternions. If the orientation is defined using an alternative representation it must undergo an initialization step that converts it to a quaternion. Appendix D outlines quaternion algebra.

Since the state description for deformable models does not explicitly specify the orientation, it simplifies to $\mathbf{s}_t = [\mathbf{p}_t, \mathbf{v}_t]$. We represent the change in state of the model as $\Delta\mathbf{s}_t = \mathbf{s}_{t+\Delta t} - \mathbf{s}_t = [\Delta\mathbf{p}_t, \Delta\mathbf{q}_t, \Delta\mathbf{v}_t, \Delta\boldsymbol{\omega}_t]$.

Let $\mathbf{N}$ denote a neural network, $\mathbf{x}$ denote a vector of neural network inputs, and $\mathbf{y}$ denote a vector of neural network outputs. The operation $\mathbf{y} = \mathbf{N}\mathbf{x}$ represents the forward pass through network $\mathbf{N}$ with input $\mathbf{x}$.

Let $\mathbf{q} = [q^r, q^x, q^y, q^z] = [r, \mathbf{v}]$ and $\mathbf{q}_1 = [q_1^r, q_1^x, q_1^y, q_1^z] = [r_1, \mathbf{v}_1]$ be two quaternion rotations. Rotation addition in quaternion space amounts to quaternion multiplication

$$\mathbf{q}\mathbf{q}_1 = rr_1 + \mathbf{v}\cdot\mathbf{v}_1 + r\mathbf{v}_1 + r_1\mathbf{v} + \mathbf{v}\times\mathbf{v}_1. \tag{4.2}$$

However, we find it is much more convenient to represent quaternion multiplication in the matrix form

$$\mathbf{q}\mathbf{q}_1 = \mathbf{Q}\mathbf{q}_1 = \begin{bmatrix} q^r & -q^x & -q^y & -q^z \\ q^x & q^r & q^z & -q^y \\ q^y & -q^z & q^r & q^x \\ q^z & q^y & -q^x & q^r \end{bmatrix} \begin{bmatrix} q_1^r \\ q_1^x \\ q_1^y \\ q_1^z \end{bmatrix}. \tag{4.3}$$

Let

$$\mathbf{R} = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} \tag{4.4}$$

denote a 3×3 rotation matrix. The operation $\mathbf{R}\mathbf{a}$ rotates vector $\mathbf{a}$ through the rotation matrix $\mathbf{R}$.

4.3.2 An Emulator That Predicts State Changes

Chapter 4 suggests training the emulator to predict a chain of evaluations of Φ that enables it to take a super timestep Δt

$$\mathbf{s}_{t+\Delta t} = \mathbf{N}_\Phi[\mathbf{s}_t, \mathbf{u}_t, \mathbf{f}_t] \tag{4.5}$$

A single sigmoid unit is nonlinear only over a small region of the input space, and it is approximately constant outside of this region. To accurately approximate a function that is

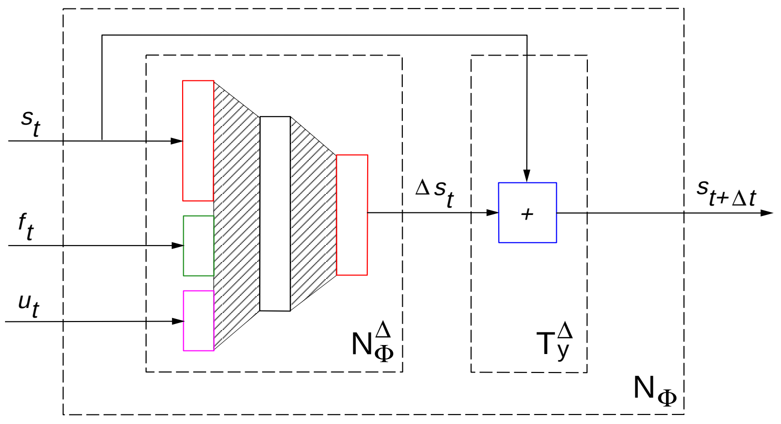


Figure 4.5: An emulator that utilizes a network $\mathbf{N}_\Phi^\Delta$ predicting the state changes. First, the emulator $\mathbf{N}_\Phi$ passes its inputs to the network $\mathbf{N}_\Phi^\Delta$ that predicts the state changes, then it computes the forward pass through this network to arrive at the vector of state changes $\Delta\mathbf{s}_t$. The last operation of the emulation step sums $\mathbf{s}_t$ and $\Delta\mathbf{s}_t$ to produce the new state $\mathbf{s}_{t+\Delta t}$ which becomes the emulator output.

nonlinear over a large domain, we need to use a neural network with multiple sigmoid units. In such a network, each sigmoid unit is shifted and scaled so that its nonlinear segment covers a different part of the domain. Clearly, the map we are trying to approximate using $\mathbf{N}_\Phi$ is nonlinear over an infinite domain. The direct approximation of this map is therefore impractical because it would require a network with an infinite number of hidden units.

To build a better approximation, we modify the network $\mathbf{N}_\Phi$ to use a subnetwork $\mathbf{N}_\Phi^\Delta$ that predicts the state changes

$$\Delta\mathbf{s}_t = \mathbf{N}_\Phi^\Delta[\mathbf{s}_t, \mathbf{u}_t, \mathbf{f}_t] \tag{4.6}$$

Fig. 4.5 shows the new architecture. First, the emulator $\mathbf{N}_\Phi$ passes its inputs to the network $\mathbf{N}_\Phi^\Delta$ that predicts the state changes, then it computes the forward pass through this network to arrive at the vector of state changes $\Delta\mathbf{s}_t$. The last operation of the emulation step sums $\mathbf{s}_t$ and $\Delta\mathbf{s}_t$ to produce the new state $\mathbf{s}_{t+\Delta t}$ which becomes the emulator output.

Clearly, the new emulator still approximates the map Φ, but it works much more accurately than the original emulator. Since the state changes are always small compared to the state values, a network that predicts the state changes has a small approximation error relative to the state value. On the other hand, a network that predicts the state values has a much larger approximation error relative to the state values which negatively impacts its accuracy. By building a network that predicts state changes we essentially eliminate a very large eigenvalue from the data.

It is easy to show the equivalence between the simple network $\mathbf{N}_\Phi$ and the emulator that utilizes the network that predicts the state changes. Let $\mathbf{x}_t^\Delta$ be the input to $\mathbf{N}_\Phi^\Delta$, and let $\mathbf{y}_t^\Delta$ be the output of $\mathbf{N}_\Phi^\Delta$. Similarly, let $\mathbf{x}_t$ be the input to $\mathbf{N}_\Phi$, and let $\mathbf{y}_t$ be the output of $\mathbf{N}_\Phi$. The following equations are true

$$\begin{aligned}
\mathbf{x}_t^\Delta &= \mathbf{x}_t \\
\mathbf{y}_t^\Delta &= \mathbf{N}_\Phi^\Delta \mathbf{x}_t^\Delta \\
\mathbf{y}_t &= \mathbf{T}_y^\Delta \mathbf{y}_t^\Delta = \mathbf{s}_t + \mathbf{y}_t^\Delta
\end{aligned}$$

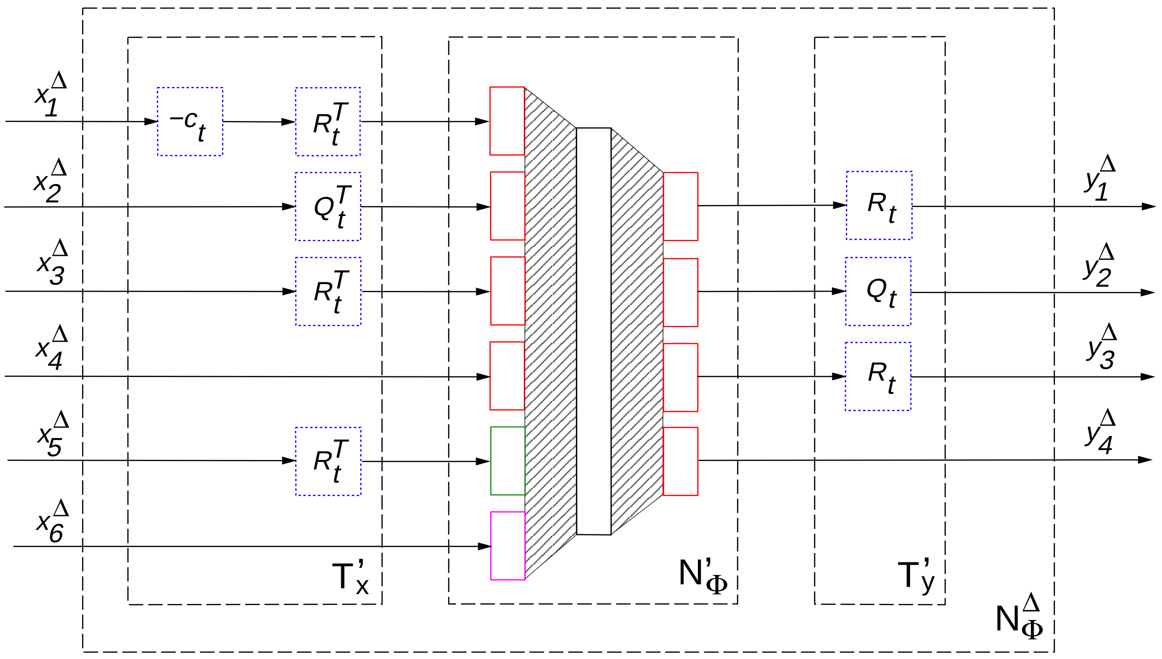


Figure 4.6: Rotation and translation invariant emulator uses translation and rotation to convert between the local and the global coordinate system. The network $\mathbf{N}'_\Phi$ operates in the local coordinate system. Dotted squares mark the subnets for which the weights change at each simulation step. See text for details.

and therefore the desired equivalence between the two emulators holds

$$\mathbf{y}_t = \mathbf{N}_\Phi \mathbf{x}_t = \mathbf{T}_y^\Delta \mathbf{N}_\Phi^\Delta \mathbf{x}_t. \tag{4.7}$$

4.3.3 Translation and Rotation Invariant Emulator

We can further improve the approximation properties of the emulator by recognizing that the map $\mathbf{\Phi}$ is invariant under rotation and translation, i.e., the state changes of a physical model do not depend on the position and orientation of the model in the global coordinate system. To this end, we build an invariant network that performs the same operation as $\mathbf{N}_\Phi^\Delta$, but uses a subnet $\mathbf{N}'_\Phi$ that operates in the local coordinate system of the model. Fig. 4.6 shows such a network. In essence, this network transforms the inputs from the global coordinate system to the coordinate system attached to the center of mass of the model through translation and rotation. It then feeds forward the localized inputs through the network $\mathbf{N}'_\Phi$ that computes the vector of localized state changes, and finally converts the output vector $\mathbf{y}'_t$ back to the global coordinate system. The details of the forward step follow.

Before the emulation step depicted in the figure begins, the system first computes the center of mass of the model $\mathbf{c}_t$, the rotation matrix $\mathbf{R}_t$ describing the orientation of the model with respect to the global coordinate system, and the quaternion rotation matrix $\mathbf{Q}_t$ that describes the same orientation as $\mathbf{R}_t$. Assuming that, at the beginning of the emulation, the local coordinate system of the emulator is aligned with the global coordinate system, matrix $\mathbf{R}_t$ rotates the body aligned with the axes of the global coordinate system to its orientation at time t, and vector $\mathbf{c}_t$ translates the center of mass of the body from the global origin to its position at time t.

Subsequently, these quantities are used to initialize the network weights responsible for the rotation and the translation, and shown in the figure as dotted squares. When the initialization step is finished, the emulator starts the forward propagation of the input signals. First comes the translation operation $-\mathbf{c}_t$ that gets applied only to the positional variables $\mathbf{p}_t$. The consecutive step applies the rotation. The reader should note that the rotation does not apply to the vector of controls $\mathbf{u}$ since its elements are scalar, and does not apply to the vector of angular velocities $\boldsymbol{\omega}_t$ since the vector is defined in the local coordinate system of the model by definition. Also, since the orientation vector $\mathbf{q}_t$ is a quaternion, it is rotated using the quaternion multiplication matrix $\mathbf{Q}_t^T$. All the other groups are multiplied by the traditional rotation matrix $\mathbf{R}_t^T$.

At this point the input vector has been aligned with the local coordinate system of the model. Next we perform the forward pass through network $\mathbf{N}'_\Phi$ which outputs the vector of state changes described in the coordinate system of the model. The last operation converts this vector back to the global coordinate system by undoing the transformations applied to the input signals of $\mathbf{N}_\Phi^\Delta$. This operation amounts to the rotation of quaternions by $\mathbf{Q}_t$, and rotation by $\mathbf{R}_t$ of all other elements. Note that the state changes should not be translated by $\mathbf{c}_t$ since they are defined in relative terms.

Our experiments show that it is much more effective to compute the map $\mathbf{N}_\Phi^\Delta$ using the network $\mathbf{N}'_\Phi$ than it is to do so directly.

We show now that the network in Fig. 4.6 is equivalent to $\mathbf{N}_\Phi^\Delta$. Let $\mathbf{x}'_t$ be the input to $\mathbf{N}'_\Phi$, and let $\mathbf{y}'_t$ be the output of $\mathbf{N}'_\Phi$, and as before, let $\mathbf{x}_t^\Delta$ denote the inputs to $\mathbf{N}_\Phi^\Delta$, and let $\mathbf{y}_t^\Delta$ denote the outputs of $\mathbf{N}_\Phi^\Delta$. The following equations are true:

$$\begin{aligned}
\mathbf{x}'_t &= \mathbf{T}'_x\mathbf{x}_t^\Delta = [\mathbf{R}_t^T(\mathbf{p}_t - \mathbf{c}_t),\ \mathbf{Q}_t^T\mathbf{q}_t,\ \mathbf{R}_t^T\mathbf{v}_t,\ \boldsymbol{\omega}_t,\ \mathbf{R}_t^T\mathbf{f}_t,\ \mathbf{u}_t] \\
\mathbf{y}'_t &= \mathbf{N}'_\Phi\mathbf{x}'_t \\
\mathbf{y}_t^\Delta &= \mathbf{T}'_y\mathbf{y}'_t = [\mathbf{R}_t\Delta\mathbf{p}'_t,\ \mathbf{Q}_t\Delta\mathbf{q}'_t,\ \mathbf{R}_t\Delta\mathbf{v}'_t,\ \Delta\boldsymbol{\omega}'_t],
\end{aligned}$$

and therefore the desired equivalence between the two networks holds

$$\mathbf{y}_t^\Delta = \mathbf{N}_\Phi^\Delta\mathbf{x}_t^\Delta = \mathbf{T}'_y\mathbf{N}'_\Phi\mathbf{T}'_x\mathbf{x}_t^\Delta. \tag{4.8}$$

4.3.4 Pre-processing and Post-processing of Training Data

Generalization performance of the network can be further improved through the pre-processing and the post-processing of the training data. In the emulation case, the pre-processing refers to a simple linear rescaling of the inputs, while the post-processing denotes the same operation applied to the outputs of the network.

As was explained earlier, the emulator inputs form groups of variables that can represent 3 different quantities: state variables, control forces, and external forces. Since the values of the variables can deviate significantly between the different groups, the impact of each group on the output of the network is not well distributed. This becomes a problem when the inputs with large sizes are relatively unimportant in determining the required outputs. Furthermore, when given unnormalized inputs with widely varying scale, multi-layer sigmoidal networks often become linear regressors.

To make each group contribute more evenly to the network output, we need to rescale its variables. Rescaling also helps initialize the weights of the network. If both inputs and outputs are normalized to have a zero mean and a unit variance, we can expect the network weights should also be of order unity. The weights can then be given a suitable random initialization prior to network training.

Although it is reasonable to assume that the variables within each group are similar, variables that belong to different groups can vary significantly and need to be treated independently. To make the variables across different groups uniform, we normalize each variable so that it has zero mean and unit variance as follows:

$$\tilde{x}_k^n = \frac{x_k^n - \mu_i^x}{\sigma_i^x}, \tag{4.9}$$

where the mean of the ith group of inputs is

$$\mu_i^x = \frac{1}{NK} \sum_{n=1}^{N} \sum_{k=k_i}^{k_i+K_i} x_k^n, \tag{4.10}$$

and its variance is

$$\sigma_i^x = \frac{1}{(N-1)(K-1)} \sum_{n=1}^{N} \sum_{k=k_i}^{k_i+K_i} (x_k^n - \mu_i)^2. \tag{4.11}$$

Here $n = 1, \ldots, N$ indexes the training example, $k = k_i, \ldots, k_i + K_i$ indexes the variables in group i, K_i represents the size of group i, and x_k^n denotes the kth input variable for the nth training example. A similar set of equations computes the means μ_j^y and the variances σ_j^y for the output layer of the network:

$$\tilde{y}_k^n = \frac{y_k^n - \mu_j^y}{\sigma_j^y}. \tag{4.12}$$

The normalization of inputs and output enables a systematic procedure for initializing weights of a network and further improves emulation quality. Fig. 4.7 illustrates such a network. The first step through this network pre-processes the inputs. This operation subtracts from each input variable the mean of the group to which the variable relates, and divides the result by the group's variance. The outcome of this operation becomes the input to $\mathbf{N}_\Phi^\sigma$. Subsequently, the emulator implements the forward pass through $\mathbf{N}_\Phi^\sigma$ generating a vector of post-processed outputs. The last step through the network undoes the effect of output post-processing. This operation multiples each output variable by the variance of the group to which the variable relates, and adds the group's mean to the result.

We now show that the network shown in Fig. 4.7 is equivalent to $\mathbf{N}'_\Phi$. Let $\mathbf{x}_t^\sigma$ be the input to $\mathbf{N}_\Phi^\sigma$, and let $\mathbf{y}_t^\sigma$ be the output of $\mathbf{N}_\Phi^\sigma$, and as before, let $\mathbf{x}'_t$ denote the inputs to $\mathbf{N}'_\Phi$, and let $\mathbf{y}'_t$ denote the outputs of $\mathbf{N}'_\Phi$. The following equations are true:

$$\begin{aligned}
\mathbf{x}_t^\sigma &= \mathbf{T}_x^\sigma \mathbf{x}'_t = [(\mathbf{x}'_1 - \mu_1^x)/\sigma_1^x, \ldots, (\mathbf{x}'_6 - \mu_6^x)/\sigma_6^x] \\
\mathbf{y}_t^\sigma &= \mathbf{N}_\Phi^\sigma \mathbf{x}_t^\sigma \\
\mathbf{y}'_t &= \mathbf{T}_y^\sigma \mathbf{y}_t^\sigma = [\mathbf{y}_1^\sigma \sigma_1^y + \mu_1^y, \ldots, \mathbf{y}_6^\sigma \sigma_6^y + \mu_6^y]
\end{aligned}$$

where $\mathbf{x}'_i$ denotes the ith input group of $\mathbf{x}'_t$, and $\mathbf{y}_i^\sigma$ denotes the ith output group of $\mathbf{y}_t^\sigma$. The above equations validate the equivalence between the two networks

$$\mathbf{y}'_t = \mathbf{N}'_\Phi \mathbf{x}'_t = \mathbf{T}_y^\sigma \mathbf{N}_\Phi^\sigma \mathbf{T}_x^\sigma \mathbf{x}'_t. \tag{4.13}$$

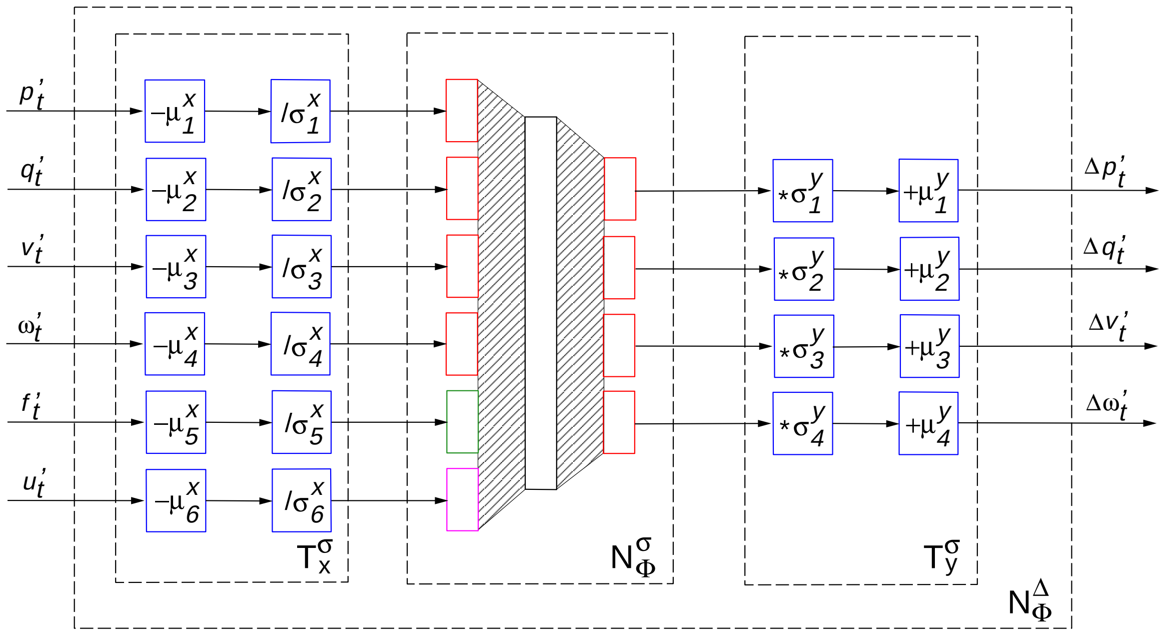


Figure 4.7: The network inputs and outputs are rescaled to have a zero mean and a unit variance to enable a better weights initialization. For each box, the operator proceeding its label denotes the operation applied to all the group's incoming signals.

4.3.5 Complete Step through Transformed Emulator

The operations outlined in this chapter form a cascade of transformations that we combine into a single emulation step, as shown in Fig. 4.4. Naturally, a NeuroAnimator that uses the cascade of transformations achieves the same functional mapping as the original network $\mathbf{N}_\Phi$. This is easy to prove using the results from the previous sections. Since (4.7) shows the equivalence between $\mathbf{N}_\Phi$ and the network in Fig. 4.5, (4.8) shows the equivalence between $\mathbf{N}^\Delta_\Phi$ and the network in Fig. 4.6, and (4.13) shows the equivalence between $\mathbf{N}'_\Phi$ and the network in Fig. 4.7, we can combine these results to prove the desired result

$$\mathbf{y}_t = \mathbf{T}^\Delta_y \mathbf{T}'_y \mathbf{T}^\sigma_y \mathbf{N}^\sigma_\Phi \mathbf{T}^\sigma_x \mathbf{T}'_x \mathbf{x}_t. \tag{4.14}$$

4.4 Hierarchical Networks

As a universal function approximator, a neural network should in principle be able to approximate the map Φ governed by (1.1) for any dynamical system, given enough sigmoid hidden units and training data. In practice, however, significant performance improvements accrue from tailoring the neural network to the physical model.

In particular, neural networks are susceptible to the curse of dimensionality. The number of neurons needed in hidden layers and the amount of training data required grows quickly with the size of the neural network, often making the training of large networks impractical. For all but the simplest mechanical models, we have found it prudent to structure the NeuroAnimator as a hierarchy of smaller networks rather than as a single large, monolithic network. The strategy behind a hierarchical state representation is to group state variables according to their dependencies and approximate each tightly coupled group with a subnet that takes part of its input from a parent net.

A natural example of hierarchical networks arises when approximating complex articulated models, such as Hodgins' mechanical human runner model (Hodgins et al., 1995) which has a tree structure with a torso and limbs. Rather than collect all the 30 controlled degrees of freedom into a single large network, it is prudent to emulate the model using 5 smaller networks: a torso network plus left and right arm and leg networks. Fig. 4.8 shows the model and the emulator networks. The torso network would include as state variables the center of mass position and orientation and its linear and angular velocity, as well as relative angles and angular velocities of the neck and waist with respect to the shoulders. Each arm network would include relative angles and angular velocities for the shoulder, elbow, and wrist joints. Each leg network includes relative angles and angular velocities for the hip, knee, ankle, and metatarsus joints.

Hierarchical representations are also useful when confronted with deformable models with large state spaces, such as the biomechanical model of a dolphin described in (Grzeszczuk and Terzopoulos, 1995) which we use in our experiments. The mass-spring dolphin model (Fig. 4.9) consists of 23 point masses, yielding a state space with a total of $23 \times 3 = 69$ positions and 69 velocities, plus 3 controlled degrees of freedom (3 actuators each consisting of 2 opposed muscle pairs). Rather than constructing a monolithic neural network with $69 + 69 = 138$ state inputs $\mathbf{s}_t$ and outputs $\mathbf{s}_{t+\Delta t}$, we subdivide hierarchically. A natural subdivision is to represent each of the 6 body segments as a separate sub-network in the local center of mass coordinates of the segment, as shown in the figure.

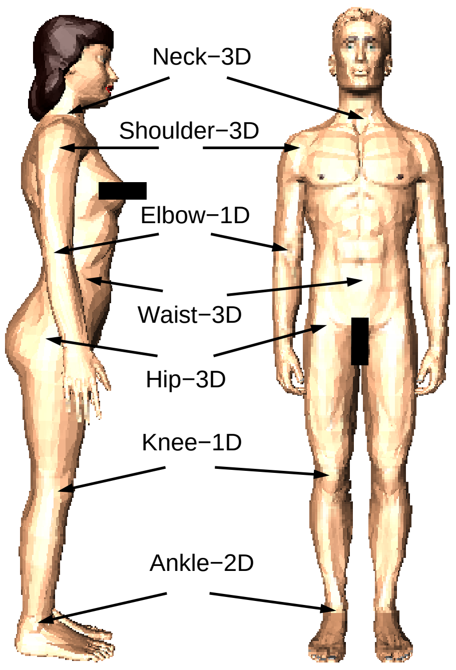
4.5 Training NeuroAnimators

This section discusses the practical issues of training a neural network to accurately approximate a dynamical simulator. First, it describes a strategy for generating independent training examples that ensure good generalization properties of the network. Subsequently, it presents the conversion step that the training data undergoes before it can be used to train the structured emulator. Finally, it lists the different optimization techniques used for neural network training.

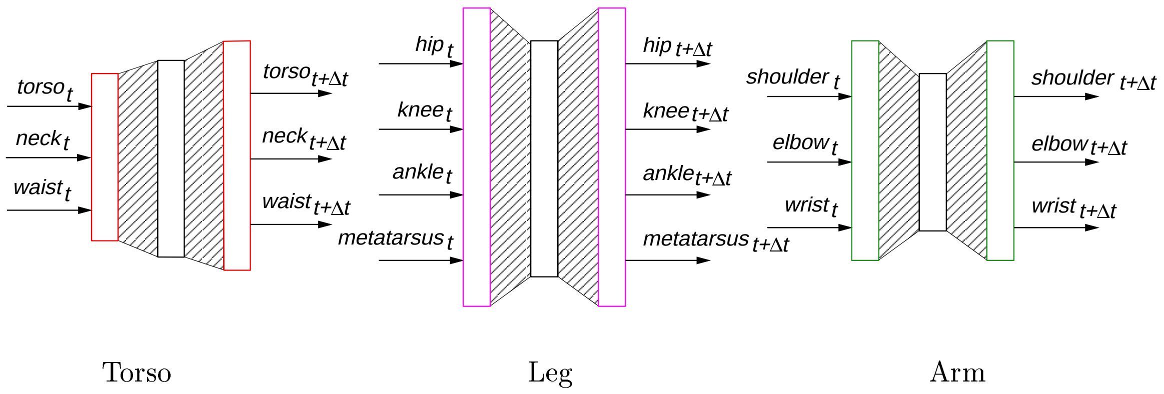
4.5.1 Training Data

To arrive at a NeuroAnimator for a given physics-based model, we train the constituent neural network(s) by invoking the backpropagation algorithm on training examples generated by simulating the model. Training requires the generation and processing of many examples, hence it is typically slow, often requiring several CPU hours. However, it is important to realize that training takes place off-line, in advance. Once a NeuroAnimator is trained, it can be reused readily to produce an infinite variety of fast animations. Training a NeuroAnimator is quite unlike recording motion capture data. In fact, the network never observes complete motion trajectories, only sparse examples of individual state transitions. The important point is that by generalizing from the sparse examples that it has learned, a trained NeuroAnimator will produce an infinite variety of extended, continuous animations that it has never seen.

More specifically, each training example consists of an input vector $\mathbf{x}$ and an output vector $\mathbf{y}$. In the general case, the input vector $\mathbf{x} = [\mathbf{s}_0^T, \mathbf{f}_0^T, \mathbf{u}_0^T]^T$ comprises the state of the model, the external forces, and the control inputs at time $t = 0$. The output vector $\mathbf{y} = \mathbf{s}_{\Delta t}$ is the state of the model at time $t = \Delta t$, where Δt is the duration of the super timestep.



(a)



(b)

Figure 4.8: (a) Human model (source: (Hodgins et al., 1995)). (b) Torso, leg, and arm emulator networks. Two arm and two leg networks are attached to a torso network to emulate the mechanical human model.

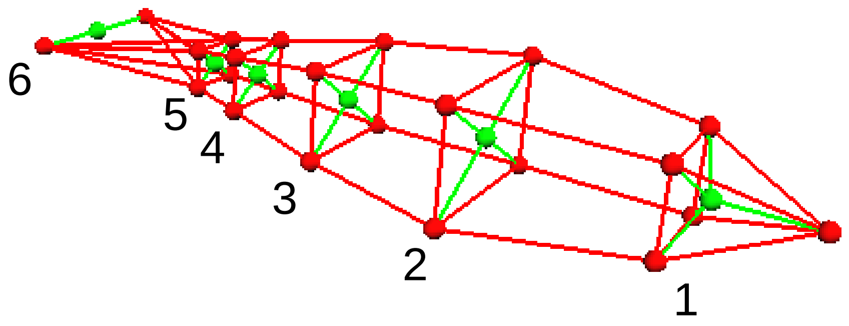


Figure 4.9: Hierarchical state representation for the dolphin mechanical model with numbered local centers marked by green (light) nodes and point masses marked by red (dark) nodes. Green lines connect point masses to associated local center.

To generate each training example, we would start the numerical simulator of the physics-based model with the initial conditions $\mathbf{s}_0$, $\mathbf{f}_0$, and $\mathbf{u}_0$, and run the dynamic simulation for n numerical time steps δt such that $\Delta t = n\delta t$. In principle, we could generate an arbitrarily large set of training examples $\{\mathbf{x}^\tau; \mathbf{y}^\tau\}$, $\tau = 1, 2, \ldots$, by repeating this process with different initial conditions.

To learn a neural network approximation $\mathbf{N}_\Phi$ of the map Φ in (1.1) for some physics-based model, it would seem sensible to sample the map by evaluating Φ as uniformly as possible over its domain. As the domain is usually of high dimensionality, this is difficult. However, we can avoid wasted effort by sampling those values of $\mathbf{s}_t$, $\mathbf{u}_t$ and $\mathbf{f}_t$ inputs that typically occur in practice. To this end, we generate the training data by uniformly sampling the subset of useful control inputs as densely as possible. That is, among the set of all possible control sequences U, we choose a subset of practical control sequences P (Fig. 4.10). We then repeatedly simulate the physical model using the controls in P generating the training examples during the simulation. This approach clusters the training data in a subset S_P of the state space S, but the system is more likely to visit a state $\mathbf{s} \in S_P$ than any state outside S_P.

Sampling of the Control Space

We generate the set P of practical control sequences using a similar control discretization approach as in (Grzeszczuk and Terzopoulos, 1995); i.e., we express the ith control function as a B-spline $u_i(t) = \sum_{i=1}^{M} u_i^j B^j(t)$, where the $B^j(t)$ are uniform B-spline basis functions, and we uniformly step each of the control points u_i^j to obtain a set of control functions which we apply to the dynamic model. Numerically simulating the motion trajectory with a fixed timestep δt, we record state, and possibly also control, and force information at

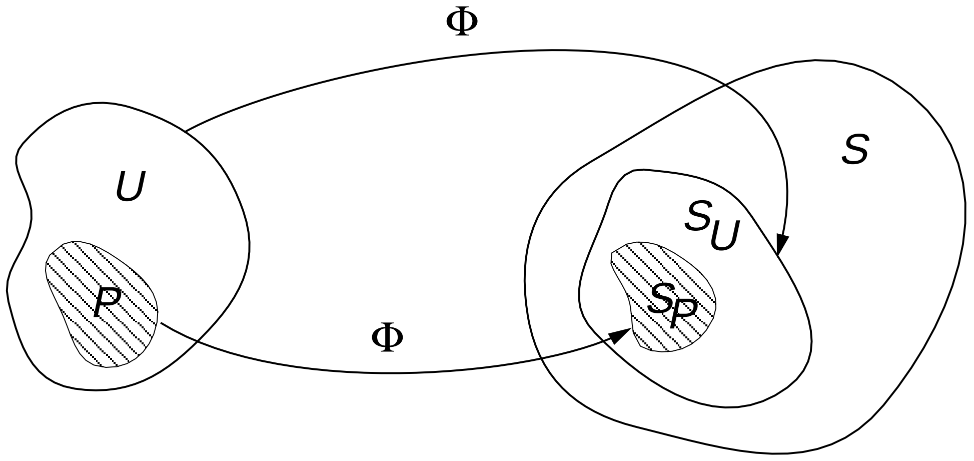


Figure 4.10: The sampling method used to explore the state space. To produce the training data we explore a subset P of the control space U that is more likely to produce a reasonable motion. We then repeatedly simulate the physical model using the controls in P generating the training examples during the simulation. This approach clusters the training data in a subset S_P of the state space S, but the system is more likely to visit a state $\mathbf{s} \in S_P$ than any state outside S_P.

t_k (where t_k denotes the k^{th} sample time), as well as state information at $t_k + \Delta t$, where $\Delta t = n\delta t$, to produce the k^{th} example.

Fig. 4.11 illustrates an effective sampling strategy using the dynamic dolphin model as an example.[1] We simulate the model over an extended period of time with a fixed timestep δt. During the simulation, we apply typical control inputs to the model. For the dolphin, the control inputs are coordinated muscle actions that produce locomotion. At well-separated times $t = t_k$ during the simulation, we record a set of training examples $\{[\mathbf{s}_{t_k}^T, \mathbf{f}_{t_k}^T, \mathbf{u}_{t_k}^T]^T; \mathbf{s}_{t_k+\Delta t}\}$, $k = 1, 2, \ldots$ The lag between successive samples is drawn randomly from a uniform distribution over the interval $\Delta t \leq (t_{k+1} - t_k) \leq 5\Delta t$. The considerable separation of successive samples in time helps reduce the correlation of the training data, improving learning. Furthermore, we randomize the order of the training samples before starting the backpropagation training algorithm. Clearly, the network observes many independent examples of typical state transitions, rather than any continuous motion.

4.5.2 Training Data For Structured NeuroAnimator

As mentioned earlier, to train the emulator shown in Fig. 4.4(c) we need only train the network $\mathbf{N}_\Phi^\sigma$ because the operators denoted by the letter $\mathbf{T}$ are predetermined. As shown in Fig. 4.12 before presenting the training data to the network $\mathbf{N}_\Phi^\sigma$, we transform the inputs of the training set through the operators $\mathbf{T}_x'$ and $\mathbf{T}_x^\sigma$ and transform the associated outputs through the operators $(\mathbf{T}_y^\Delta)^{-1}$, $(\mathbf{T}_y')^{-1}$, and $(\mathbf{T}_\Phi^\sigma)^{-1}$ which are the inverses of the corresponding operators used during the forward emulation step shown in Fig. 4.4(c).

[1]The reader should note that the data used to train the runner emulator had not been generated as described in this section. The method used to produce the training data for this particular problem is described in the results chapter.

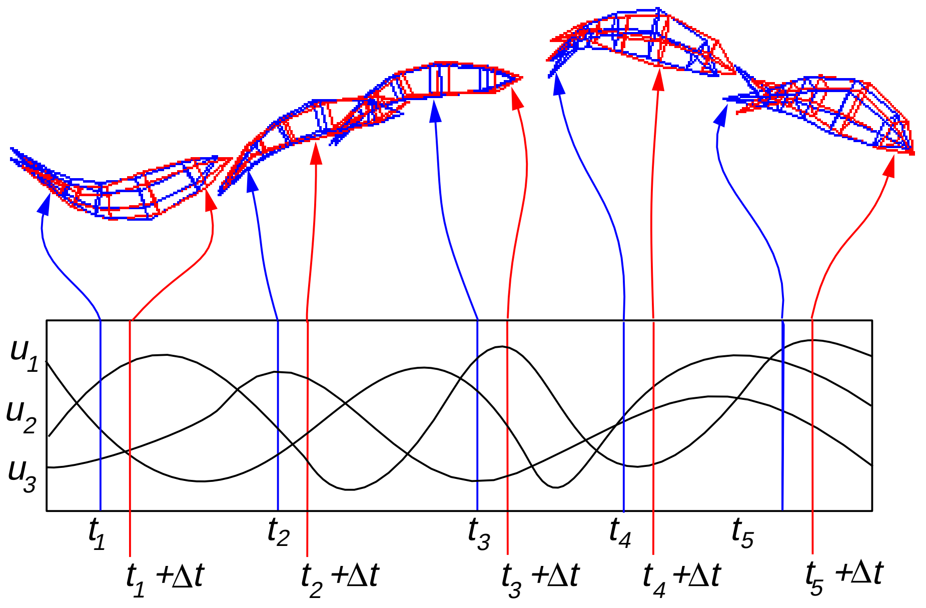


Figure 4.11: An effective state transition sampling strategy illustrated using the dynamic dolphin model. The dynamic model is simulated numerically with typical control input functions **u**. For each training example generated, the blue model represents the input state (and/or control and external forces) at time t_k, while the red model represents the output state at time $t_k + \Delta t$. The long time lag enforced between samples reduces the correlation of the training examples that are produced.

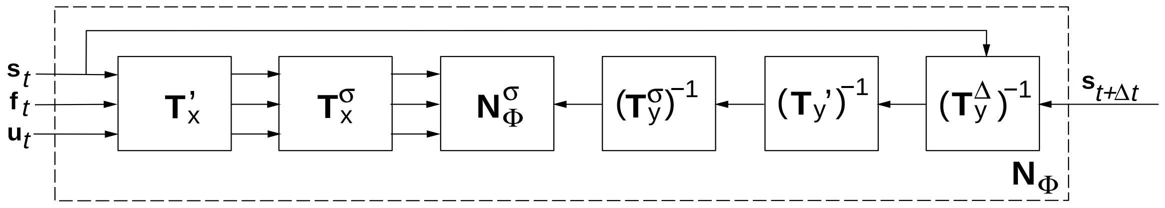


Figure 4.12: Transforming the training data for consumption by the network $\mathbf{N}^\sigma_\Phi$ in Fig. 4.4(c). The inputs of the training set are transformed through the operators on the input side of the network in Fig. 4.4(c). The outputs of the training set are transformed through the inverses of the operators at the output side of the network in Fig. 4.4(c).

4.5.3 Network Training in Xerion

We begin the off-line training process by initializing the weights of $\mathbf{N}_{\Phi}^{\sigma}$ to random values from a uniform distribution in the range $[0, 1]$ (due to the normalization of inputs and outputs). Xerion automatically terminates the backpropagation learning algorithm when it can no longer reduce the network approximation error (3.2) significantly.

We use the conjugate gradient method to train networks of small and moderate size. This method converges faster than gradient descent, but the efficiency becomes less significant when training large networks. Since this technique works in batch mode, as the number of training examples grows, the weight updates become too time consuming. For this reason, we use gradient descent with the momentum term (Section 3.4.2) when training large networks. We divide the training examples into small sets, called *mini-batches*, each consisting of approximately 30 uncorrelated examples, and update the network weights after processing each mini-batch.

Appendix F contains an example Xerion script which specifies and trains a NeuroAnimator.

Chapter 5

NeuroAnimator Controller Synthesis

In this chapter we turn to the problem of control; i.e., producing physically realistic animation that satisfies goals specified by the animator. We first describe the objective function and its discrete approximation and then propose an efficient gradient based optimization procedure that computes derivatives of the objective function with respect to the control inputs through the back-propagation algorithm.

5.1 Motivation

A popular approach to the animation control problem is *controller synthesis* (van de Panne and Fiume, 1993; Ngo and Marks, 1993; Grzeszczuk and Terzopoulos, 1995). Controller synthesis is a generate-and-test strategy. Through repeated forward simulation of the physics-based model, it optimizes a control objective function that measures the degree to which the animation generated by the controlled physical model achieves the desired goals. Each simulation is followed by an evaluation of the motion through the function, thus guiding the search.

While the controller synthesis technique readily handles the complex optimal control problems characteristic of physics-based animation, it is computationally very costly. Evaluation of the objective function requires a forward dynamic simulation of the dynamic model, often subject to complex applied forces and constraints. Hence the function is almost never analytically differentiable, prompting the application of non-gradient optimization methods such as simulated annealing (van de Panne and Fiume, 1993; Grzeszczuk and Terzopoulos, 1995) and genetic algorithms (Ngo and Marks, 1993). In general, since gradient-free optimization methods perform essentially a random walk through the huge search space of possible controllers, computing many dynamic simulations before finding a good solution, they generally converge slowly compared to methods that are guided by gradient directions. Although slow, this technique is very robust and can be applied to complex computer models with little effort. Fig. 5.1 shows some of the models of animals from (Grzeszczuk and Terzopoulos, 1995) that have used motion synthesis to successfully learn actuator coordination necessary for locomotion.

The NeuroAnimator enables a novel, highly efficient approach to controller synthesis. One reason for the efficiency of this new approach is the fast emulation of the dynamics of the physical model. But the main reason for the efficiency is that we can exploit the

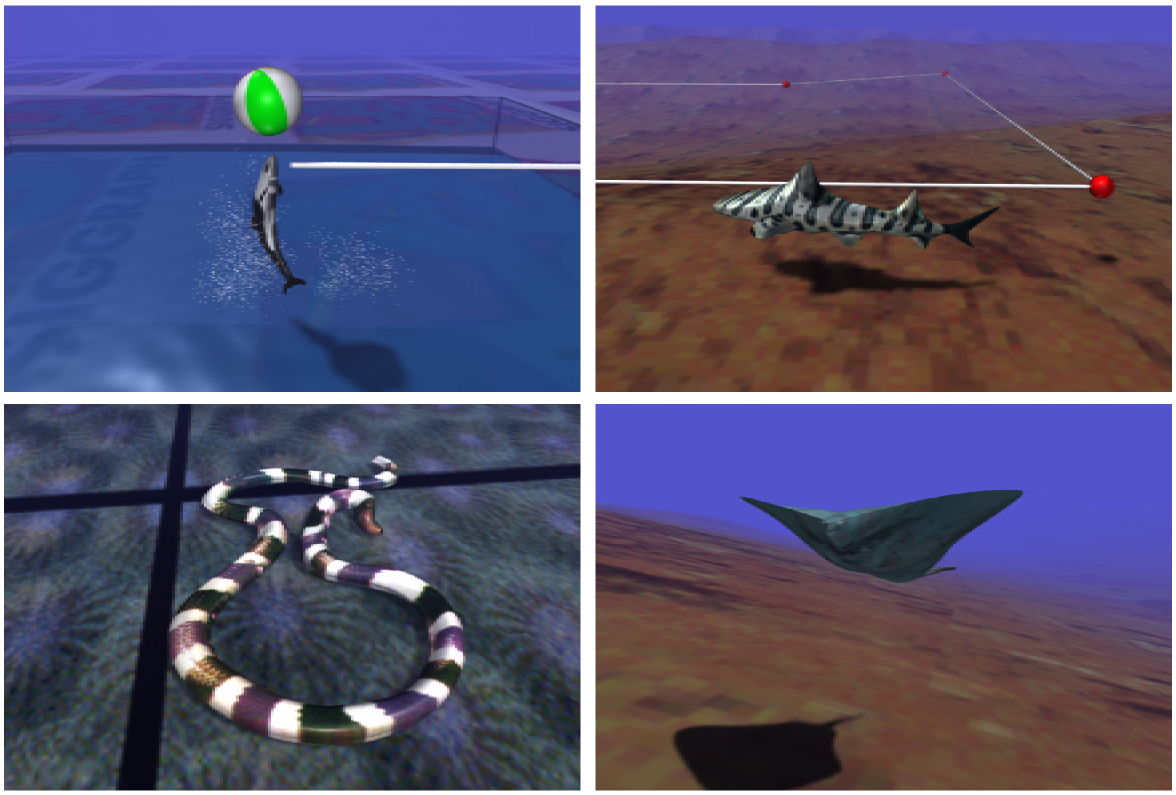


Figure 5.1: Complexity of some models that have used motion synthesis to successfully learn actuator coordination necessary for locomotion.

neural network approximation in the trained NeuroAnimator to compute partial derivatives of output states with respect to control inputs. This enables the computation of a gradient, hence the use of fast gradient-based optimization for controller synthesis.

In the remainder of this chapter, we first describe the objective function and its discrete approximation. We then propose an efficient gradient based optimization procedure that computes derivatives of the objective function with respect to the control inputs through a back-propagation algorithm.

5.2 Objective Function and Optimization

Using (4.1) we write a sequence of emulation steps

$$\mathbf{s}_{i+1} = \mathbf{N}_\Phi[\mathbf{s}_i, \mathbf{u}_i, \mathbf{f}_i]; \qquad 1 \leq i \leq M \tag{5.1}$$

where i indexes the emulation step, and $\mathbf{s}_i$, $\mathbf{u}_i$ and $\mathbf{f}_i$ denote, respectively, the state, control inputs and external forces in the ith step. Fig. 4.1 illustrates forward emulation by the NeuroAnimator according to this index notation.

Following the control learning formulation in (Grzeszczuk and Terzopoulos, 1995), we define a discrete objective function

$$J(\mathbf{u}) = \mu_u J_u(\mathbf{u}) + \mu_s J_s(\mathbf{s}), \tag{5.2}$$

a weighted sum (with scalar weights μ_u and μ_s) of a term J_u that evaluates the controller $\mathbf{u} = [\mathbf{u}_1, \mathbf{u}_2, \ldots, \mathbf{u}_M]$ and a term J_s that evaluates the motion $\mathbf{s} = [\mathbf{s}_1, \mathbf{s}_2, \ldots, \mathbf{s}_{M+1}]$ produced by the network model using that controller, according to (5.1) during a time interval $t_0 \leq t \leq t_M$. We may wish to promote a preference for controllers with certain desirable qualities, such as smooth lower amplitude controllers, via the controller evaluation term J_u. The distinction between good and bad control functions also depends on the goals that the animation must satisfy. In our applications, we used trajectory criteria J_s such as the final distance to the goal, the deviation from a desired speed, etc. The objective function provides a quantitative measure of the progress of the controller learning process, with larger values of J indicating better controllers.

A typical objective function used in our experiments seeks an efficient controller that leaves the model in some desired state $\mathbf{s}_d$ at the end of simulation. Mathematically, this is expressed as

$$J(\mathbf{u}) = \frac{\mu_u}{2} \sum_{i=1}^{M} \mathbf{u}_i^2 + \frac{\mu_s}{2} (\mathbf{s}_{M+1} - \mathbf{s}_d)^2, \tag{5.3}$$

where the first term maximizes the efficiency of the controller and the second term constrains the final state of the model at the end of the animation.

5.3 Backpropagation Through Time

Assuming a trained NeuroAnimator with a set of fixed weights, the essence of our control learning algorithm is to iteratively update the control parameters $\mathbf{u}$ so as to maximize the objective function J in (5.2). As mentioned earlier, we exploit the NeuroAnimator structure

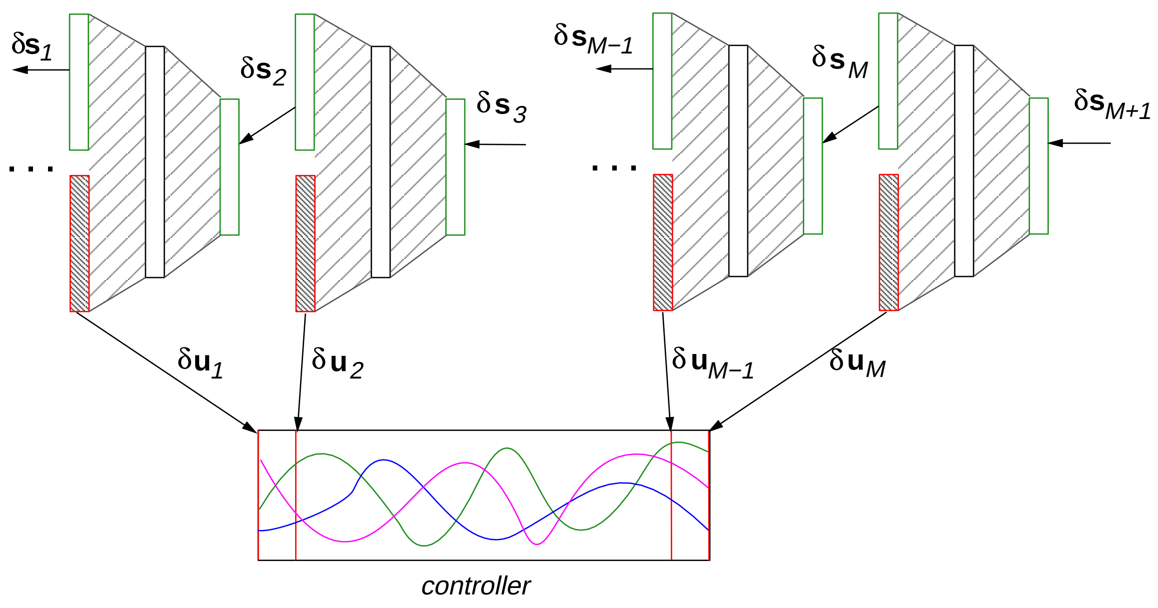


Figure 5.2: The backpropagation through time algorithm. At each iteration the algorithm computes the derivatives of the objective function with respect to the inputs of the emulator using the chain rule and it adjusts the control inputs to decrease the value of the objective function.

to arrive at an efficient gradient ascent optimizer:

$$\mathbf{u}^{l+1} = \mathbf{u}^l + \eta_x \nabla_{\mathbf{u}} J(\mathbf{u}^l), \tag{5.4}$$

where l denotes the iteration of the minimization step, and constant η_x is the learning rate parameter used during the input update.

At each iteration l, the algorithm first emulates the forward dynamics according to (5.1) using the control inputs $\mathbf{u}^l$ to yield the motion sequence $\mathbf{s}^l$, as is illustrated in Fig. 4.1. Next, the algorithm computes the components of $\nabla_{\mathbf{u}} J$ in an efficient manner. The cascade network structure enables us to apply the chain rule of differentiation within each network and backwards across networks, yielding a variant of the backpropagation algorithm called *backpropagation through time*. Instead of adjusting weights as in normal backpropagation, however, the algorithm adjusts neuronal inputs as presented in Section 3.3.2. It thus proceeds in reverse through the network cascade computing components of the gradient. Fig. 5.2 illustrates the backpropagation through time process, showing the sequentially computed controller updates $\delta\mathbf{u}_M$ to $\delta\mathbf{u}_0$. See Appendix C for details on an efficient, recursive implementation of the input adjustment algorithm.

The forward emulation and control adjustment steps are repeated for each iteration of (5.4), quickly yielding a good controller. The efficiency stems from two factors. First, each NeuroAnimator emulation of the physics-based model consumes only a fraction of the time it would take to numerically simulate the model. Second, quick gradient ascent towards an optimum is possible because the trained NeuroAnimator provides a gradient direction. An additional advantage of the approach is that once an optimal controller has been computed, it can be applied to control either the NeuroAnimator emulator or to the original physical model, yielding animations that in most cases differ only minimally.

The control algorithm based on the differentiation of the emulator of the forward model has very important advantages. First, the backpropagation through time can solve fairly complex sequential decision problems where early decisions can have substantial effects on the final results. Second, the algorithm can be applied to dynamic environments with changing control objectives since it relearns very quickly. The system is therefore much more adaptive than reinforcement learning where each change in the environment requires complete policy reevaluation—a tremendously expensive computation.

5.3.1 Momentum Term

More efficient optimization techniques can be applied to improve a slow convergence rate of the gradient descent algorithm used in (5.4). Adding the momentum term (Section 3.4.2) to the gradient descent rule improves the effective learning rate of the update rule

$$\delta \mathbf{u}^{l+1} = \eta_x \nabla_{\mathbf{u}} J(\mathbf{u}^l) + \alpha_x \delta \mathbf{u}^l \tag{5.5}$$

$$\mathbf{u}^{l+1} = \mathbf{u}^l + \delta \mathbf{u}^{l+1} \tag{5.6}$$

where α_x is the momentum parameter used to update the inputs, and l is the iteration of the minimization step.

Learning with the momentum term is very fast and takes only a few iterations to solve problems that would otherwise have taken much longer. Section 6.2 includes the performance comparison for the different optimization techniques.

5.3.2 Control Learning in Hierarchical NeuroAnimators

In Section 4.4 we presented hierarchically defined emulators, that combine a set of networks trained on different motion aspects of the model for more economic representation. The hierarchical emulator has, in general, one network that represents the global aspects of motion, and a set of networks that refine motion produced by the global network. We have shown that the hierarchical emulators are faster to train and emulate more efficiently. But the hierarchical emulators make control learning easier as well. In the case of the dolphin emulator, for example, we use the global deformation network only during the control learning since the local deformations of the model do not impact the result. Similarly for the human model, when learning a controller that uses a subset of joints, we only need to evaluate the networks that represent this subset.

5.3.3 Extending the Algorithm to Arbitrary Objectives

Until now we have assumed that the objective is a function defined in terms of the states and the controls of the model as in (5.2). Although this definition covers a wide range of practical problems, the objective does not have to be such a function, but can be represented instead by a scalar value V. Since we do not know the dependence of V on the states and the controls, we need to find a mapping $f(\cdot)$ that defines the relation

$$V = f(\mathbf{u}_1, \ldots, \mathbf{u}_M, \mathbf{s}_1, \ldots, \mathbf{s}_{M+1}) = f(\mathbf{u}, \mathbf{s}). \tag{5.7}$$

The influence of the control inputs on the map $f(\cdot)$ can be both direct—if $f(\cdot)$ contains the controller evaluation term—and indirect since the control inputs determine the state changes. The latter association is much harder to discover, but fortunately it is already embedded in the emulator. By making $\mathbf{N}_\Phi$ a part of $f(\cdot)$, we can learn the objective function quickly from a small number of examples.

Notice that the situation discussed here resembles reinforcement learning where the system needs to construct a critic that evaluates the model's performance according to some control objective. In general, building the critic is a tough problem that requires many evaluations of the objective. However, we can combine the emulator $\mathbf{N}_\Phi$ with the map $f(\cdot)$ that can be learned rather quickly, to form the critic with significantly less effort.

5.4 Control Adjustment Step Through Structured NeuroAnimator

Section 4.3 introduced a NeuroAnimator that has been structured to approximate a physical model very accurately. Fig. 4.4 shows the schematic representation of this emulator. The input signals of this emulator pass through a sequence of standard transformations before they reach the principal network $\mathbf{N}^\sigma_\Phi$. The forward pass through the main network produces a set of inputs, which then undergo a second set of transformations.

The control learning algorithm computes the gradient of the objective function $\mathbf{J}$ with respect to the inputs of the structured emulator for each emulation step. The algorithm starts by evaluating the derivatives with respect to the last emulation step, and then proceeds to the earlier steps, accumulating the results along the way. This was explained in Section 5.3. In this section we discuss the details of a single backpropagation step through the structured emulator.

The backpropagation step starts at the output layer of the emulator that has been initialized by the derivative computation for the previous emulation step. The algorithm proceeds towards the inputs of the emulator, computing recursively the derivatives of $\mathbf{J}$ with respect to each layer of units.

Note that there is a close relationship between the forward pass and the backward pass through the NeuroAnimator. During the forward pass, the relation between the two neighboring layers can be written down as

$$y_j = g(\sum_{i=1}^{p} x_i w_{ij}). \tag{5.8}$$

A similar relation holds between the derivatives in the neighboring layers of units during the backward pass that we have derived in Appendix C:

$$\delta x_i = \sum_{j=1}^{q} \delta y_j w_{ij} g'(y_j), \tag{5.9}$$

where $g'(\cdot)$ denotes the derivative of the activation function, δy_j is the derivative with respect to the output j, and y_j is the activation of this unit. Equation (5.9) simplifies even further in the case when $g'(\cdot)$ is the identity function

$$\delta x_i = \sum_{j=1}^{q} \delta y_j w_{ij}. \tag{5.10}$$

The last relation leads to some interesting observations that shed light into the imple-

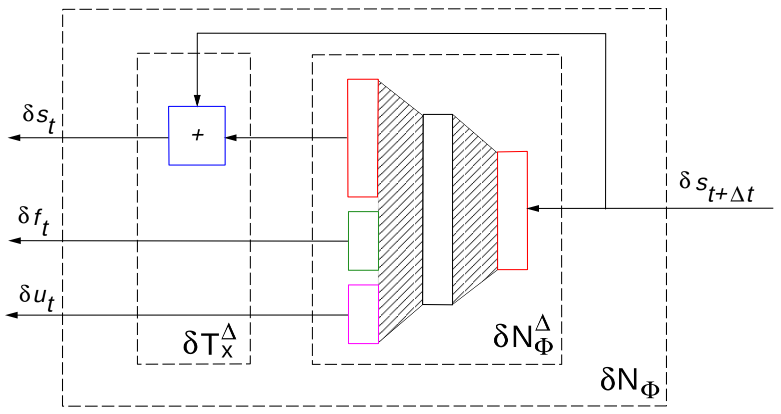

δs_t
δf_t
δu_t
+
$\delta \mathrm{T}^{\Delta}_{x}$
$\delta \mathrm{N}^{\Delta}_{\Phi}$
$\delta \mathrm{N}_{\Phi}$
$\delta s_{t+\Delta t}$

Figure 5.3: The backpropagation step through the emulator that utilizes a network predicting state changes.

mentation of the derivative computation for the NeuroAnimator. Mainly, if $y = sx$, then $\delta x = s\delta y$. This means that if during the forward pass we scale signal x by s, then during the backward pass we also need to scale signal δy by s. If, on the other hand, $\mathbf{y} = \mathbf{Rx}$ where $\mathbf{R}$ is a rotation matrix, then $\delta\mathbf{x} = R^T\delta\mathbf{y}$. This fact is easy to show since according to (5.8)

$$y_j = \sum_{i=1}^{3} x_i \mathbf{r}_i.$$

where $\mathbf{r}_i$ is the ith row of R, and according to (5.10)

$$\delta x_i = \sum_{j=1}^{3} \delta y_j \mathbf{r}_j$$

where $\mathbf{r}_j$ is the jth column of R. A similar relationship holds between the forward and the backward pass for the quaternion rotation matrix $\mathbf{Q}$, i.e., $\mathbf{y} = \mathbf{Qx}$ during the forward pass becomes $\delta\mathbf{x} = Q^T\delta\mathbf{y}$ during the backward pass.

Figures 5.3–5.5 use the above relations to show the details of the backpropagation step through each transformations introduced in Section 4.3, while Fig. 5.6 puts all these transformations into a single backpropagation step. The reader should note that the hidden layer of the network $\mathbf{N}^{\sigma}_{\Phi}$ is the only layer of sigmoid units in the whole emulator. During the backpropagation through this layer, the delta signals get multiplied by the derivative of the objective functions.

5.5 Example

We will highlight some aspects of the control learning algorithm using a simple example. Suppose we want to find an efficient controller that leaves the dynamical system in some desired state $\mathbf{s}_d$. We can express this objective mathematically as

$$J = -\frac{1}{2}(\mathbf{s}_d - \mathbf{s}_{M+1})^2 - \frac{1}{2}\sum_{i=1}^{M}\mathbf{u}_i^2 = J_s + J_u \tag{5.11}$$

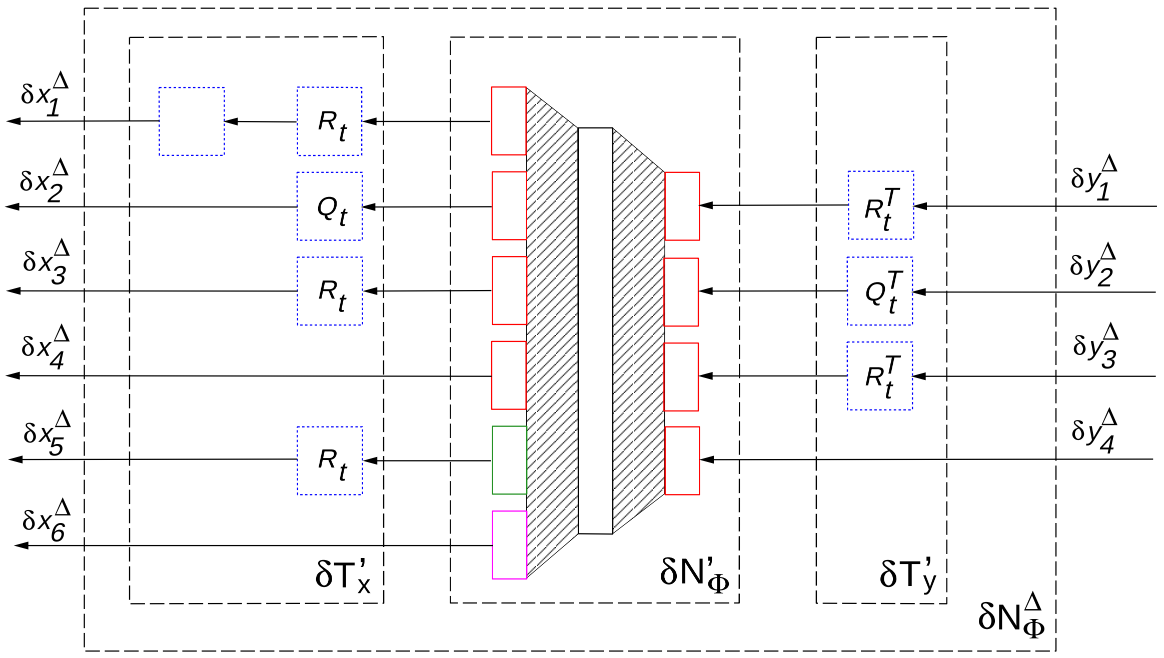


Figure 5.4: The backpropagation step through the rotation and translation invariant emulator. Note that the order of rotations is reversed and there is no translation.

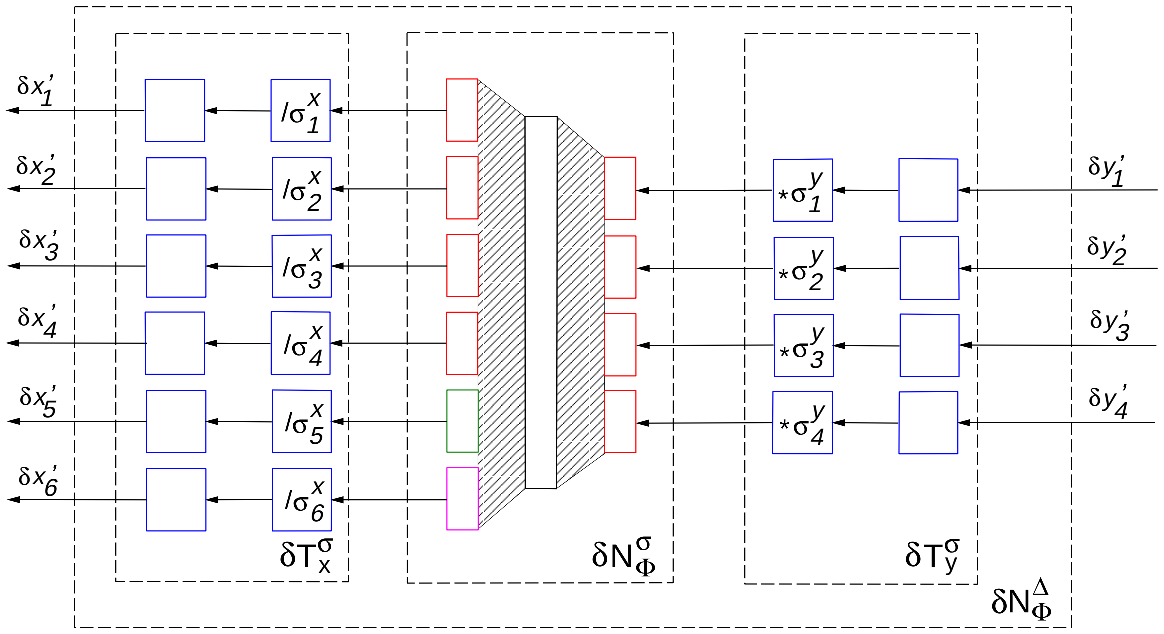


Figure 5.5: The backpropagation step through the network with inputs and outputs are rescaled to have zero mean and unit variance for better weights initialization. Note that during the backpropagation step the scaling factors remain the same as during the forward step, but the translations disappear.

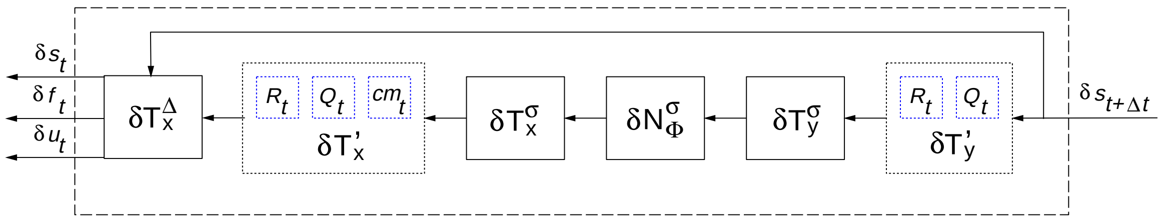


Figure 5.6: The backpropagation step through the structured emulator. The process first applies a set of transformations to the network outputs, then performs the backpropagation step through the main subnetwork of the emulator $\mathbf{N}_{\Phi}^{\sigma}$, and finally applies a second set of transformation to the subnetwork inputs. The emulator structure has been designed to compute the error derivatives with maximum accuracy.

where J_s is the term responsible for the proper configuration of the model at the end of the simulation, and J_u is the term that maximizes the efficiency of the controller.

The derivative of J_s with respect to output $\mathbf{s}_{M+1}$ has a simple form

$$\frac{\partial J_s}{\partial \mathbf{s}_{M+1}} = \mathbf{s}_d - \mathbf{s}_{M-1}. \tag{5.12}$$

Using the chain rule and (5.12) we get the expression for the derivative of J_s with respect to the input vector $\mathbf{s}_M$

$$\frac{\partial J_s}{\partial \mathbf{s}_M} = (\mathbf{s}_d - \mathbf{s}_{M-1})\frac{\partial \mathbf{s}_{M+1}}{\partial \mathbf{s}_M}. \tag{5.13}$$

Similarly, we can get the expression for the derivative of J_s with respect to the input vector $\mathbf{u}_M$

$$\frac{\partial J_s}{\partial \mathbf{u}_M} = (\mathbf{s}_d - \mathbf{s}_{M-1})\frac{\partial \mathbf{s}_{M+1}}{\partial \mathbf{u}_M}. \tag{5.14}$$

Appendix C shows the detailed computation of the partials occurring above. Since J_u does not depend on the state, we have that

$$\frac{\partial J}{\partial \mathbf{s}_M} = \frac{\partial J_s}{\partial \mathbf{s}_M} \tag{5.15}$$

and

$$\frac{\partial J}{\partial \mathbf{u}_M} = \frac{\partial J_s}{\partial \mathbf{u}_M} - \mathbf{u}_M. \tag{5.16}$$

We have now computed the derivatives of the objective function with respect to the input layer at the last iteration of the emulation. We can use the result to iteratively compute the derivatives with respects to the inputs at the earlier iterations. Once we have all the derivatives ready, we can adjust the control inputs using the on-line input update rule (3.8)

$$\mathbf{u}_i^{l+1} = \mathbf{u}_i^l - \eta_x \frac{\partial J}{\partial \mathbf{u}_i^l}. \tag{5.17}$$

Chapter 6

Results

This chapter presents a set of trained NeuroAnimators and supplies performance benchmarks and an error analysis for them. Additionally, the chapter discusses the use of the regularization step during the emulation of nonrigid models and its impact on the approximation error. The second part of the chapter describes the results of applying the new control learning algorithm to the trained NeuroAnimators. The report includes the comparison of the new technique with the control learning techniques used previously in computer graphics literature.

6.1 Example NeuroAnimations

We have successfully used the methodology described in the previous chapter to construct and train several NeuroAnimators to emulate a variety of physics-based models, including the 3-link pendulum from Fig. 4.2, a lunar lander spacecraft, a truck, a dolphin, and a runner as pictured in the figures that follow. We used SD/FAST[1] to simulate the dynamics of the rigid body and articulated models, and we employ the simulator developed in (Tu and Terzopoulos, 1994) to simulate the dynamics of the deformable-body dolphin. Table 6.1 summarizes the structure of the NeuroAnimators emulating these models in our experiments (note that for the hierarchical dolphin model of Fig. 4.9 we indicate the dimensions for one of the networks in the bottom layer, since the others are similar). In our experiments we have not attempted to minimize the number of network weights required for successful training. We have also not tried to minimize the number of hidden units, but rather used enough to obtain networks that generalize well while not overfitting the training data. We can always expect to be able to satisfy these guidelines in view of our ability to generate sufficient training data. Section 6 will present a detailed analysis of our results, including performance benchmarks indicating that the neural network emulators can yield physically realistic animation one or two orders of magnitude faster than conventional numerical simulation.

The runner NeuroAnimator differs from the other NeuroAnimator in that it can produce only a single motion sequence, and is not a complete emulator of the model dynamics. This limitation is due to the sparseness of the training data that we had available for the model. The emulator used a 30 sec. simulation sequence produced at $\delta t = 0.0027$ sec.

[1]SD/FAST is a commercial system for simulating rigid body dynamics, available from Symbolic Dynamics, Inc.

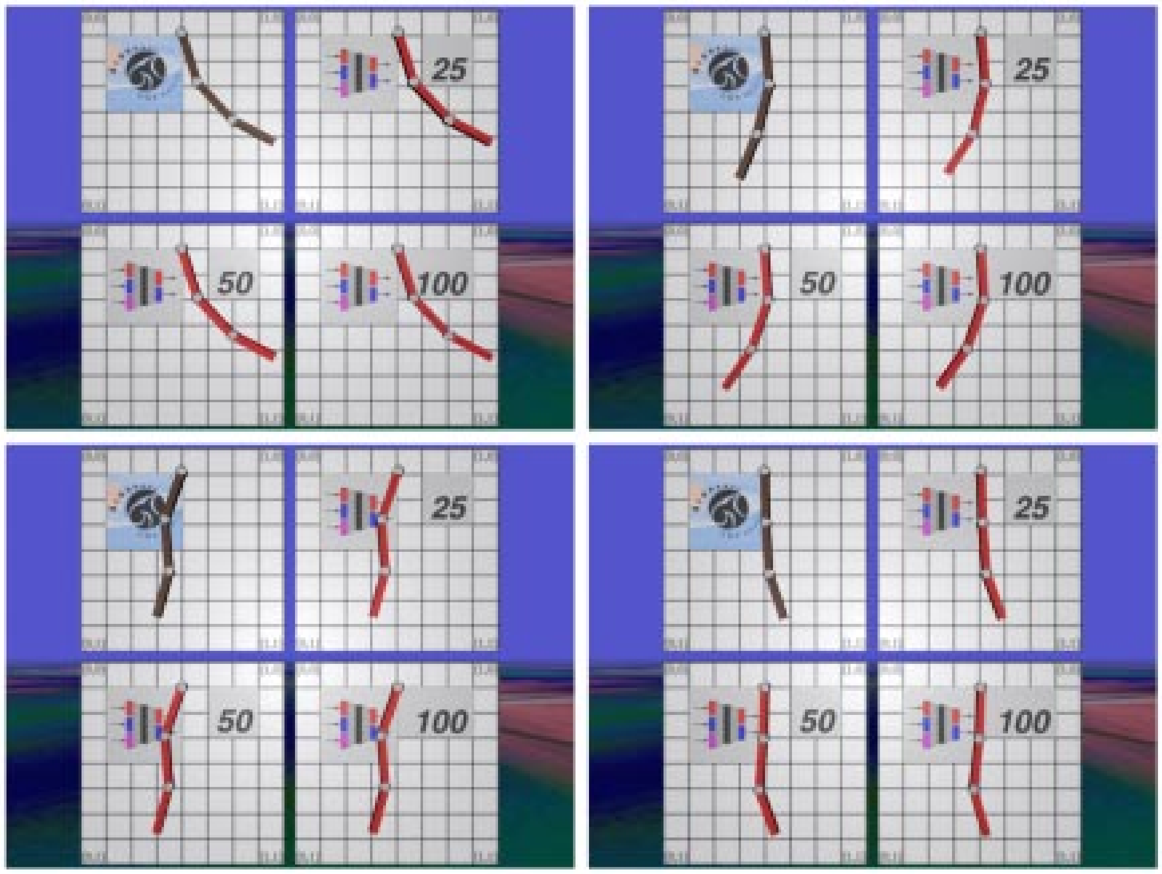


Figure 6.1: The emulation of a simple dynamical system. The upper left display in each of the 4 panels shows a numerically simulated physical model of a three-link pendulum swinging freely in gravity. The other displays show NeuroAnimators faithfully emulating the physical model. Each NeuroAnimator was trained using super timesteps corresponding to 25, 50, and 100 simulator timesteps.

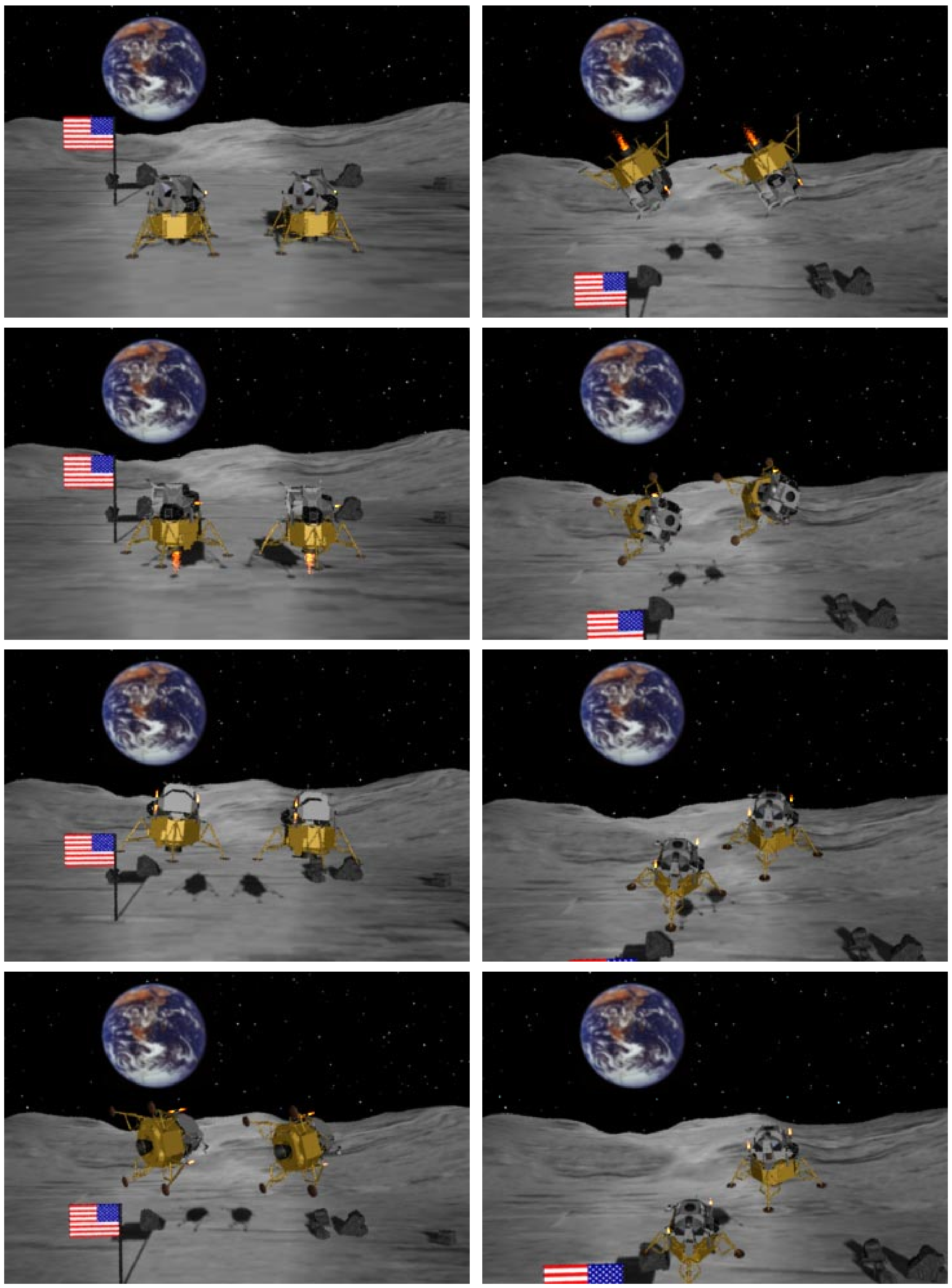


Figure 6.2: Frames from the animation comparing the response of the physical system and its emulator to the same set of control inputs for the case of the space ship. The model on the left represents the physical system, the model on the right—its emulator.

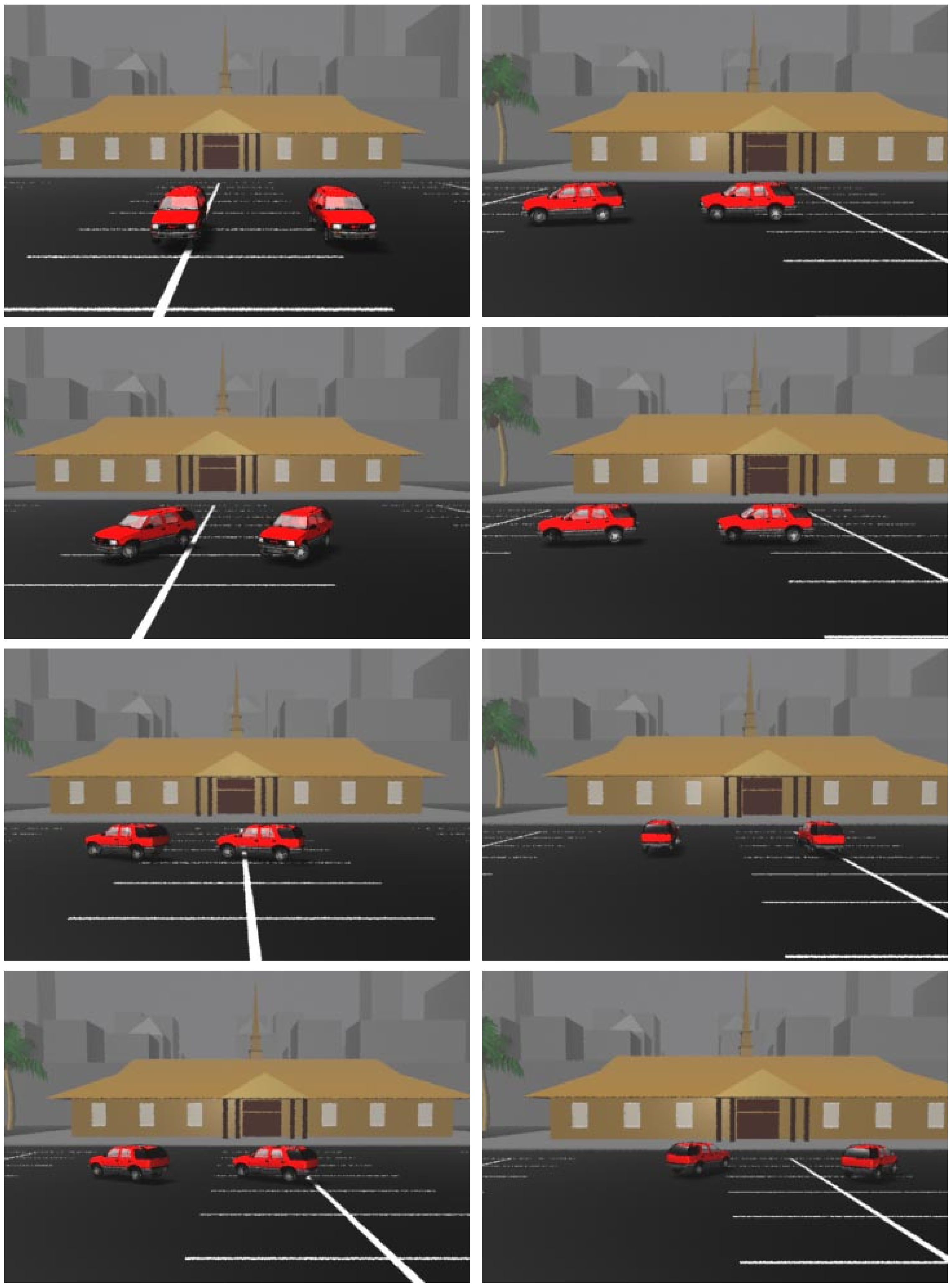


Figure 6.3: Frames from the animation comparing the response of the physical system and its emulator to the same set of control inputs for the case of the truck. The model on the left represents the physical system, the model on the right—its emulator.

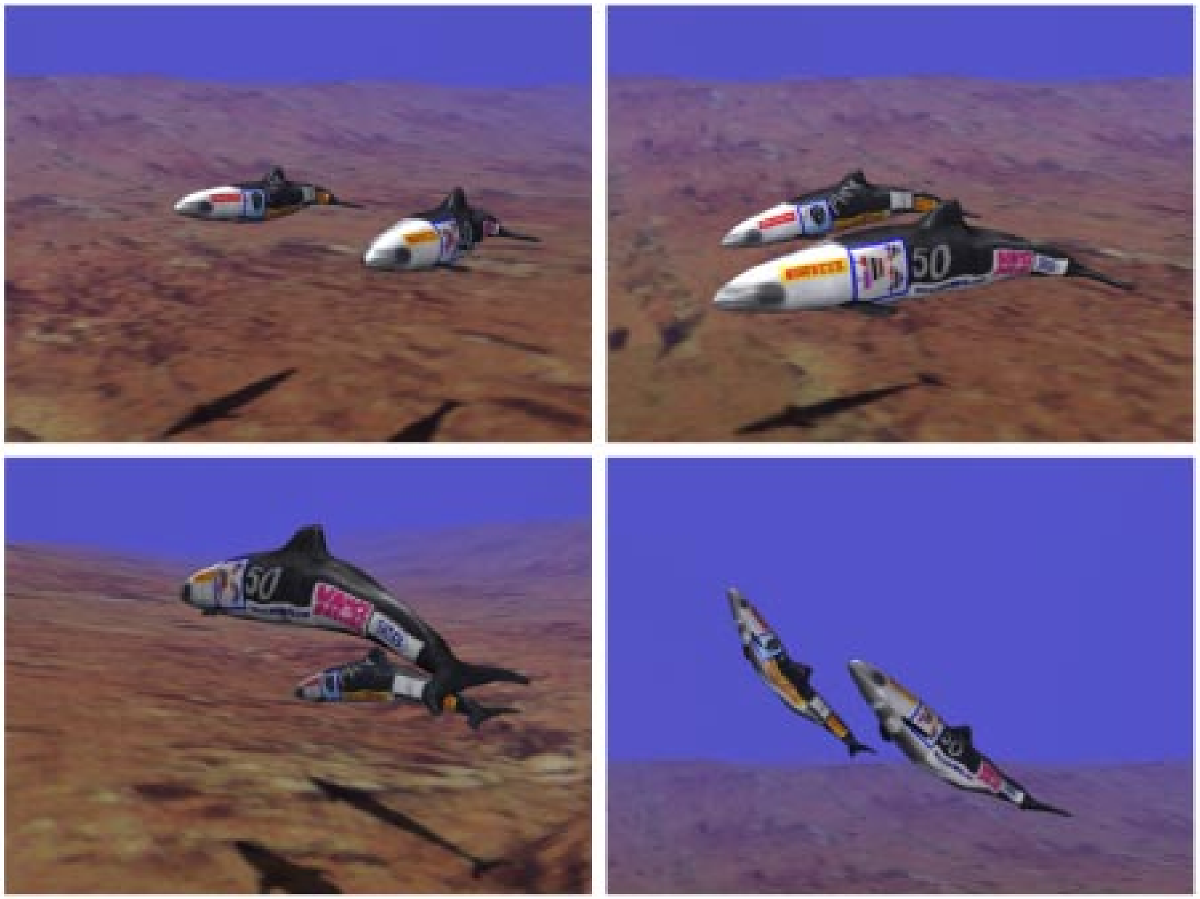


Figure 6.4: The far dolphin is animated by simulating its biomechanical model which is actuated by internal muscle actions. The near dolphin is animated by a NeuroAnimator that emulates the physical model at much less computational cost.

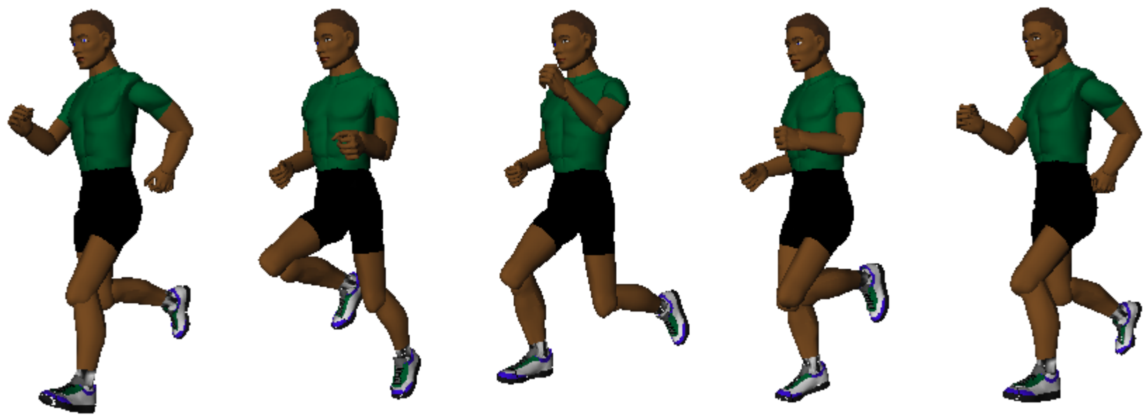


Figure 6.5: Emulated motion produced by training a NeuroAnimator on state transition data generated by the Hodgins mechanical runner model, simulated numerically by SD/FAST. The trained neural network emulator produces this running animation. The data used for training the runner was a 30 sec. simulation sequence produced at $\delta t = 0.0027$ sec. using SD/FAST. From this sequence we generated 11,000 training examples to train the networks for the runner model.

Model Description	State Inputs	Force Inputs	Control Inputs	Hidden Units	State Outputs	Training Examples
Pendulum						
passive	6	—	—	20	6	2,400
active	6	—	3	20	6	3,000
ext. force	6	3	3	20	6	3,000
Lander	13	—	4	50	13	13,500
Truck	6	—	2	40	6	5,200
Dolphin						
global net	78	—	6	50	78	64,000
local net	72	—	6	40	36	32,000
Runner						
torso	23	—	—	30	23	14,000
arm	12	—	—	30	12	7,200
leg	12	—	—	30	12	7,200

Table 6.1: Structure of the NeuroAnimators used in our experiments. Columns 2, 3, and 4 indicate the input groups of the emulator, column 4 indicates the number of hidden units, and column 5 indicates the number of outputs. The final column shows the size of the data set used to train the model. The dolphin NeuroAnimator includes six local nets, one for each body segment.

Model Description	Physical Simulation	$\mathbf{N}_\Phi^{25}$	$\mathbf{N}_\Phi^{50}$	$\mathbf{N}_\Phi^{100}$	$\mathbf{N}_\Phi^{50}$ with Regularization
Passive Pendulum	4.70	0.10	0.05	0.02	—
Active Pendulum	4.52	0.12	0.06	0.03	—
Truck	4.88	—	0.07	—	—
Lunar Lander	6.44	—	0.12	—	—
Dolphin	63.00	—	0.95	—	2.48

Table 6.2: Comparison of simulation time between the physical simulator and different neural network emulators. The duration of each test was 20,000 physical simulation timesteps.

using SD/FAST to generated 11,000 training examples to train the networks for the runner model.

For each experiment, to ensure an unbiased evaluation of the generalization quality of the emulators, we use the test data that comprises examples that were not part of the training set.

6.1.1 Performance Benchmarks

An important advantage of using neural networks to emulate dynamical systems is the speed at which they can be iterated to produce animation. Since the emulator for a dynamical system with the state vector of size N never uses more than $O(N)$ hidden units, it can be evaluated using only $O(N^2)$ operations. Appendix A contains the computer code for the forward step. By comparison, a single simulation timestep using an implicit time integration scheme requires $O(N^3)$ operations.[2] Moreover, a forward pass through the neural network is often equivalent to as many as 50 or 100 physical simulation steps, so the acceleration is even more dramatic, yielding performance improvements as great as 100 times faster than the physical simulator.

In the remainder of this section we use $\mathbf{N}_\Phi^n$ to denote a neural network model that was trained with super timestep $\Delta t = n\delta t$. Table 6.1.1 compares the physical simulation times obtained using the SD/FAST physical simulator and 3 different neural network models: $\mathbf{N}_\Phi^{25}$, $\mathbf{N}_\Phi^{50}$, and $\mathbf{N}_\Phi^{100}$. The neural network model that predicts over 100 physical simulation steps offers a speedup of anywhere between 50 and 100 times depending on the type of the system. In case of the deformable dolphin model, the first column indicates the simulation time using the physical simulator described in (Tu and Terzopoulos, 1994), the third column show the simulation time using $\mathbf{N}_\Phi^{50}$ emulator, and the last column shows the impact of regularization on the emulation time. In this case, each emulation step includes 5 iteration of the regularizer described in Section 6.1.3. For the car model and the lunar lander model we have trained only the $\mathbf{N}_\Phi^{50}$ emulators.

6.1.2 Approximation Error

An interesting property of the neural network emulation, as Fig. 6.6 testifies, is that the error does not increase appreciably for emulators with increasingly larger super timesteps;

[2]Strictly speaking, the implicit numerical methods take $O(N^3)$ operations only if all the variables of the physics-based model depend on each other. Most often, however, the physics-based models are sparsely connected. If banded matrix techniques are used, then the numerical methods take $O(p^2N)$, where p is the bandwidth of the matrix. For non-banded systems, (preconditioned) conjugate gradient can be used. If each variable is connected to k others, then conjugate gradient converges in $O(kN^2)$.

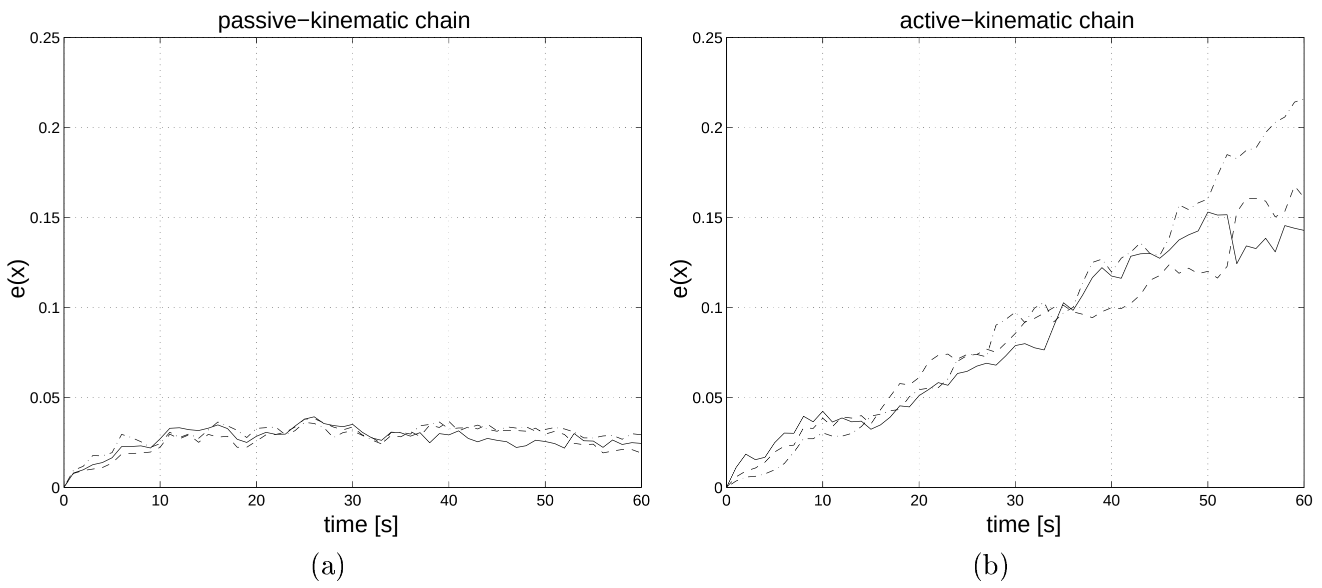


Figure 6.6: The error $e(\mathbf{x})$ in the state estimation incurred by different neural network emulators, measured as the absolute difference between the state variables of the emulator and the associated physical model. Plot (a) compares the approximation error for the passive pendulum for 3 different emulator networks: $\mathbf{N}_{\Phi}^{25}$ (solid), $\mathbf{N}_{\Phi}^{50}$ (dashed), $\mathbf{N}_{\Phi}^{100}$ (dot-dashed). Plot (b) shows the same comparison for the active pendulum. In all experiments, we averaged the error over 30 simulation trials and over all state variables. The duration of each trial was 6000 physical simulation timesteps.

i.e., in the graphs, the error over time for $\mathbf{N}_{\Phi}^{25}$, $\mathbf{N}_{\Phi}^{50}$, and $\mathbf{N}_{\Phi}^{100}$ is nearly the same. This is attributable to the fact that an emulator network that can predict further into the future must be iterated fewer steps per given interval of animation than must an emulator that can't predict so far ahead. Thus, although the error per iteration may be higher for the longer-range emulator, the growth of the error over time can remain nearly the same for both the longer and shorter range predictors. This means that the only penalty for using emulators that predict far ahead might be a loss of detail (high frequency components in the motion) due to coarse sampling. However, we did not observe this effect for the physical models with which we experimented. This suggests that the physical systems are locally smooth. Of course, it is not possible to increase the neural network prediction time indefinitely, because eventually the network will no longer be able to approximate the physical system at all adequately.

Although it is hard to totally eliminate error, we noticed that for the purposes of computer animation the approximation error remained within reasonable bounds. The neural network emulation appears comparable to the physical simulation, and although the emulated trajectory differs slightly from the trajectory produced by the physical simulator, the emulator seems to reproduce all of the visually salient properties of the physical motion.

6.1.3 Regularization of Deformable Models

When emulating spring-mass systems in which the degrees of freedom are subject to soft constraints, we discovered that the modest approximation error of even a well-trained emulator network can accumulate as the network is applied repeatedly to generate a lengthy animation. Unlike an articulated system whose state is represented by joint angles and hence is kinematically constrained to maintain its connectivity, the emulation of mass-spring systems can result in some unnatural deformations after many (hundreds or thousands of) emulation steps. Accumulated error can be annihilated by periodically performing regularization steps through application of the true dynamical system (1.1) using an explicit Euler time integration step

$$\begin{aligned} \mathbf{v}_{t+\delta t} &= \mathbf{v}_t + \delta t f(\mathbf{s}_t), \\ \mathbf{x}_{t+\delta t} &= \mathbf{x}_t + \delta t \mathbf{v}_{t+\delta t}, \end{aligned}$$

where the state is $\mathbf{s}_t = [\mathbf{v}_t, \mathbf{x}_t]^T$ and $f(\mathbf{s}_t)$ is the spring deformation force at time t. It is important to note that this inexpensive, explicit Euler step is adequate as a regularizer, but it is impractical for long-term physical simulation because of its instability. In order to apply the explicit Euler step we used a smaller spring stiffness and larger damping factor when compared to the semi-explicit Euler step used during the numerical simulation (Tu and Terzopoulos, 1994). Otherwise the system would oscillate too much or would simply become unstable.

We achieve the best results when performing a small number of the regularization steps after each emulation step. This produces much smoother motion than performing more regularization step but less frequently. To measure the deformation of the model we define the *deformation energy*

$$E_d = \frac{1}{2}\sum_{i=0}^{m}\left[\frac{l_i^c - l_i^r}{l_i^r}\right]^2$$

where m is the number of springs, l_i^c is the current length of spring i, and l_i^r is its rest length. Fig. 6.7 shows how the regularization reduces the deformation energy of the model. The solid line shows the deformation energy of the emulator that follows each emulation step with 5 regularization steps. This is enough to keep the deformation approximately constant and roughly at the same level as during the physical simulation. When the emulator does not use the regularizer, the deformation energy continuously increases with time.

6.2 Control Learning Results

This section presents results of applying the new control learning algorithm to the trained NeuroAnimators. The report includes the comparison of the new technique with the control learning techniques used previously in the computer graphics literature.

We have successfully applied our backpropagation through time controller learning algorithm to the trained NeuroAnimators. We find the technique very effective—it routinely computes solutions to non-trivial control problems in just a few iterations.

For example, a swimming controller for the dolphin model was synthesized in just 20 learning emulations. The convergence rate is breathtaking compared with control synthesis techniques that do not exploit gradient information (van de Panne and Fiume, 1993; Grzeszczuk and Terzopoulos, 1995). In particular, (Grzeszczuk and Terzopoulos, 1995) reports that the same swimming controller took 3000 physical simulations to synthesize using simulated annealing, and 700 iterations using the simplex method. The efficiency accrued from our fast convergence rate is further amplified by the replacement of costly physical simulation with much faster NeuroAnimator emulation. These two factors yield enormous

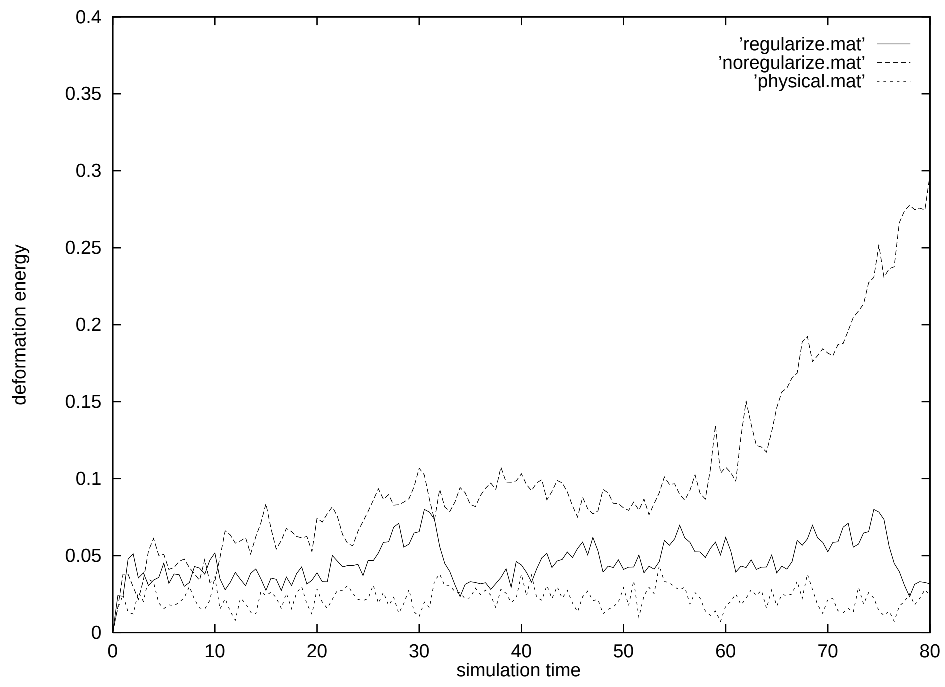


Figure 6.7: The plot compares deformation energy for 3 different cases: emulation with regularization, emulation without regularization, and physical simulation. The first case performs 5 regularization steps after each emulation step. This is enough to keep the deformation energy at the same level as during the physical simulation.

speedups—the synthesis of the swimming controller which took more than 1 hour using the technique in (Grzeszczuk and Terzopoulos, 1995) now takes less than 10 seconds on the same computer.

Fig. 6.8 shows the progress of the control learning algorithm for the 3-link pendulum. The purple pendulum, animated by a NeuroAnimator, has a goal to end the simulation with zero velocity in the position indicated by the green pendulum. We make the learning problem very challenging by setting a low upper limit on the internal motor torques of the pendulum, so that it cannot reach its target in one shot, but must swing back and forth to gain the momentum necessary to reach the goal state. Our algorithm takes 20 iterations to learn a controller that successfully achieves the goal.

Fig. 6.9 shows the lunar lander NeuroAnimator learning a soft landing maneuver. The translucent lander resting on the surface indicates the desired position and orientation of the model at the end of the animation. An additional constraint is that the downward velocity upon landing should be small enough to land softly. A successful landing controller was computed in 15 learning iterations.

Fig. 6.10 shows the truck NeuroAnimator learning to park. The translucent truck in the background indicates the desired position and orientation of the model at the end of the simulation. The NeuroAnimator produces a parking controller in 15 learning iterations.

Fig. 6.11 shows the dolphin NeuroAnimator learning to swim forward. Simple objective of moving as far forward as possible produces a natural looking, sinusoidal swimming pattern.

All trained controllers are 20 seconds long, i.e., they take 2,000 physical simulation timesteps, or 40 emulator super timesteps using $\mathbf{N}_{\Phi}^{50}$ emulator. The number of control

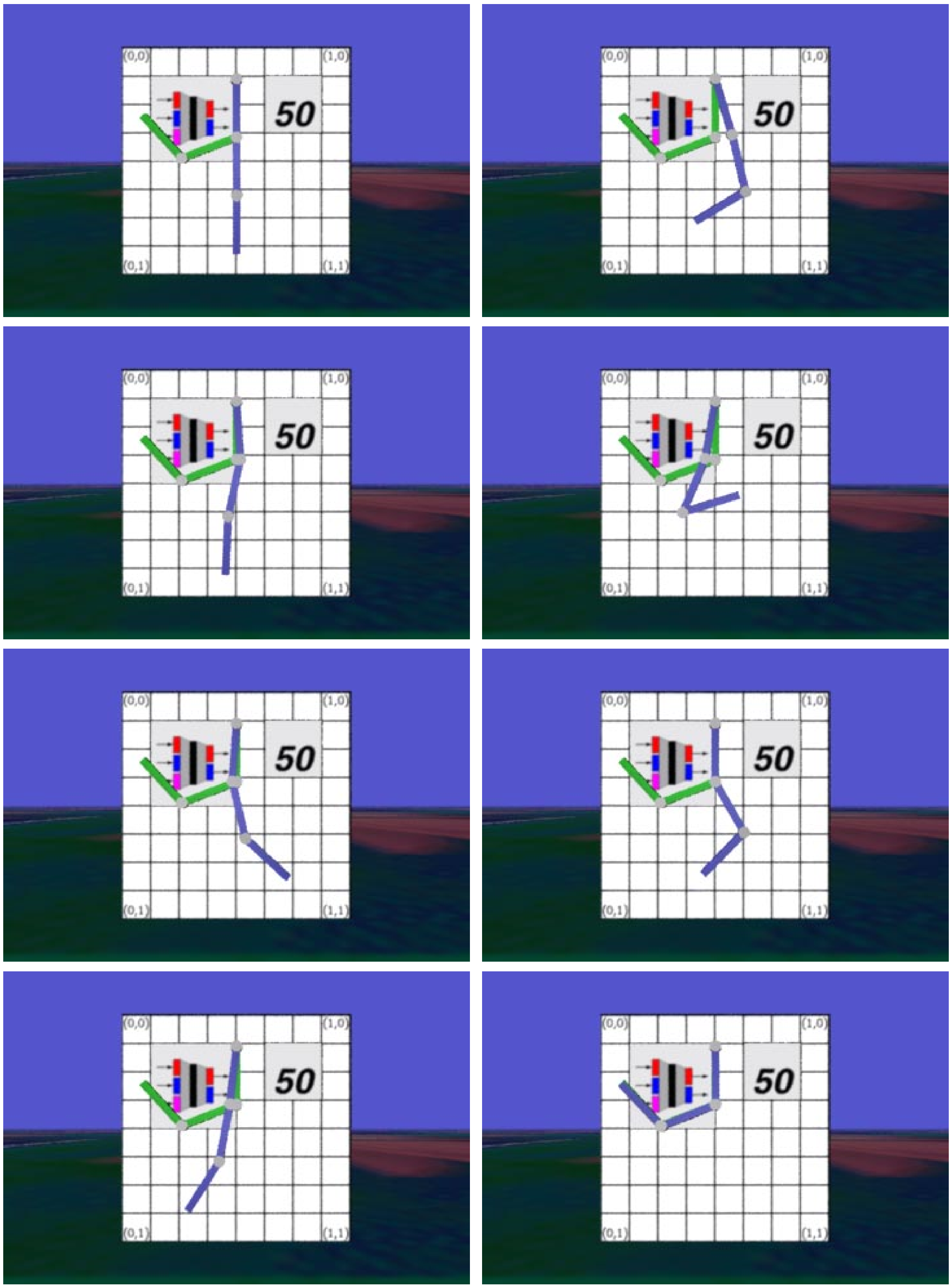


Figure 6.8: Frames from the animation showing progress of the control learning algorithm for the 3-link pendulum. The purple pendulum representing the emulator has a goal to end the simulation in the position indicated by the green pendulum at zero velocity.

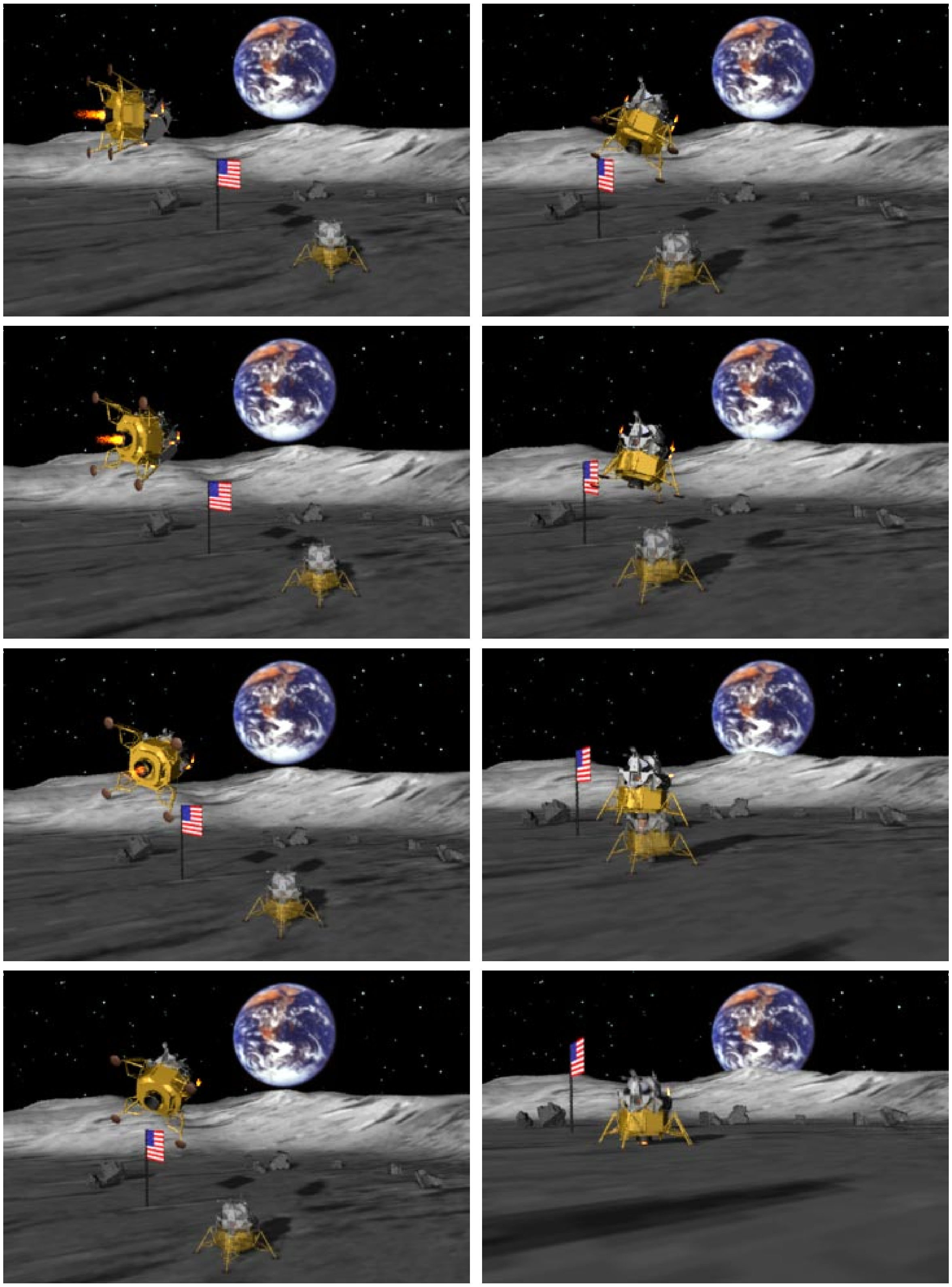


Figure 6.9: Frames from the animation showing progress of the control learning algorithm for the space ship. The model resting on the surface of the planet indicates the desired position and orientation of the emulator at the end of the simulation. Minimal velocity at landing constitutes an additional requirement.

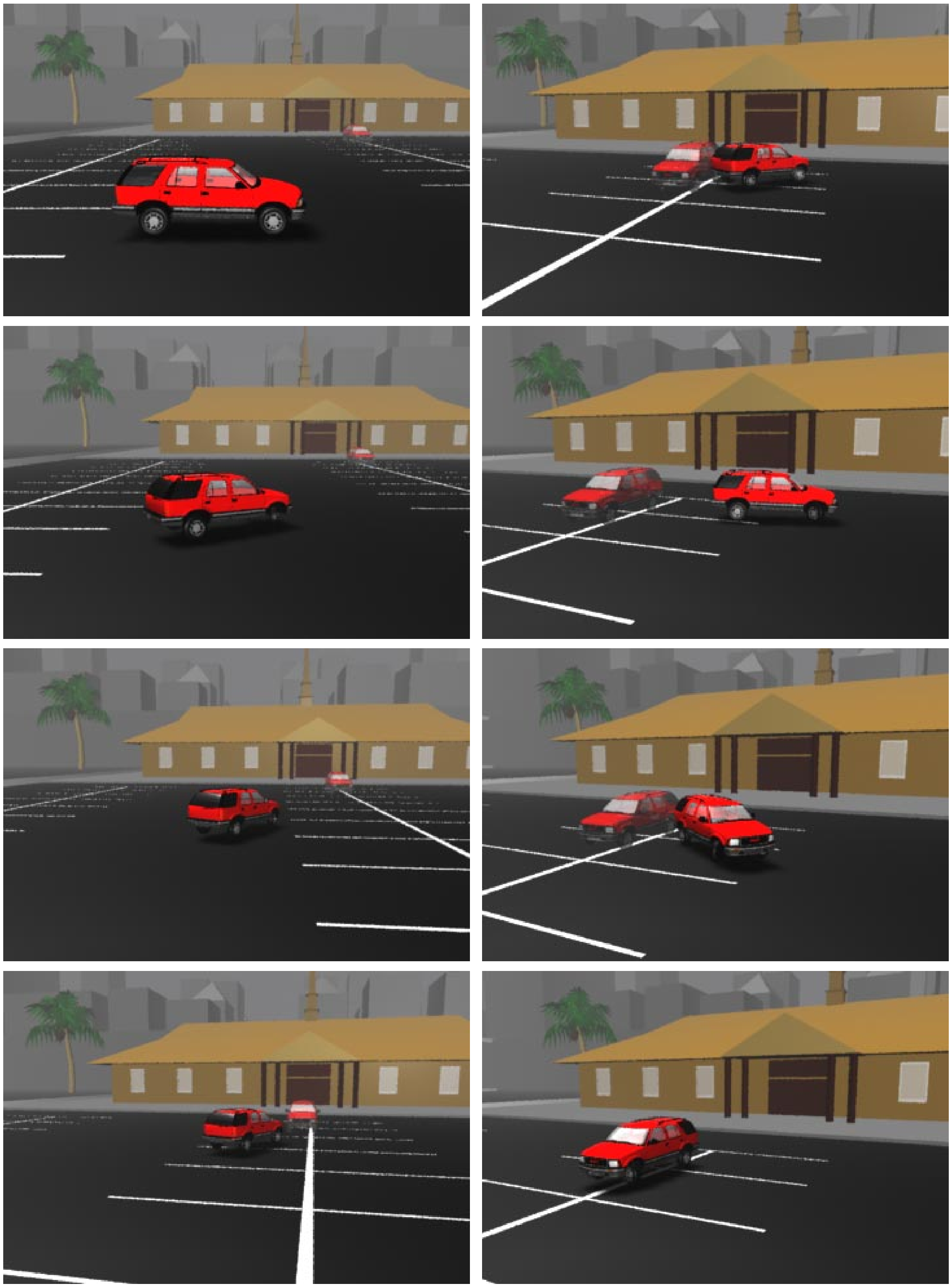


Figure 6.10: Frames from the animation showing the truck learning to park. The model far in the background indicates the desired position and orientation of the emulator at the end of the simulation.

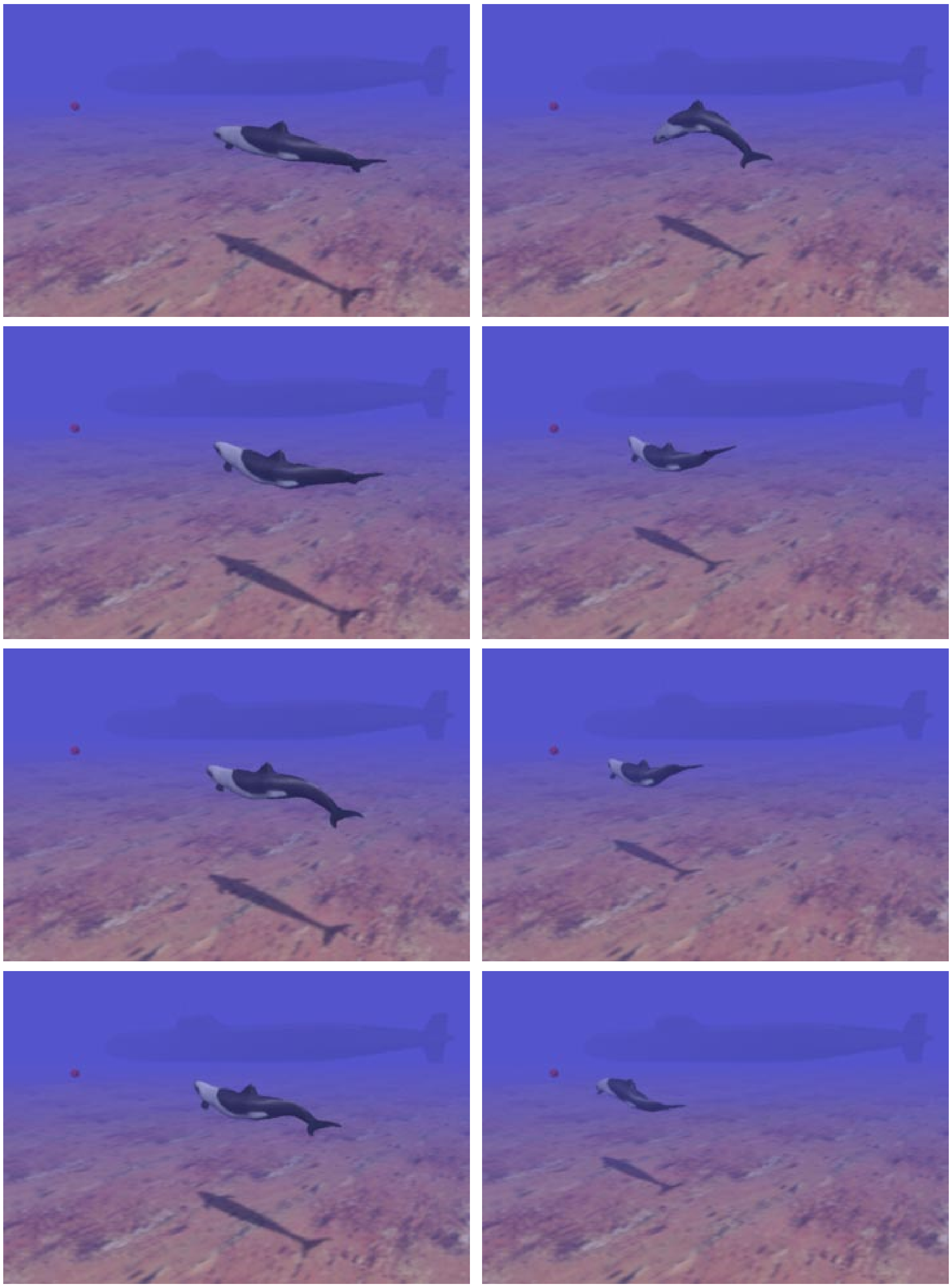


Figure 6.11: Frames from the animation showing the dolphin learning to swim. Simple objective of moving as far forward as possible produces a natural looking, sinusoidal swimming pattern.

variables optimized varies: the pendulum optimizes 60 variables, 20 for each actuator; the lunar lander optimizes 80 variables, 20 for the main thruster, and 20 for each of the 3 attitude thrusters; the car optimizes 40 variables—20 for acceleration/deceleration, and 20 for the rate of turning; finally, the dolphin optimizes 60 variables—one variable for every 2 emulator steps for each of the six muscles.

6.3 Directed vs Undirected Search in Control Space

This section compares the efficiency of the gradient directed control learning algorithm proposed in this thesis with the efficiency of the undirected search techniques presented in (Grzeszczuk and Terzopoulos, 1995).

We use the locomotion learning problem studied in (Grzeszczuk and Terzopoulos, 1995) as the test case for our experiment. In this problem, the dolphin model needs to learn how to actuate its 6 independent muscles over time in order to swim forward efficiently. The task is defined in terms of the objective function that measures the distance of the model from some far away goal. Fig. 6.11 shows the frames from the animation depicting the learning process.

Fig. 6.12 reports the results of the experiment for the different algorithm as plots of the progress of learning as a function of the number of iterations. The plot on the left has been taken from (Grzeszczuk and Terzopoulos, 1995) and shows the progress of learning using two undirected search methods: simulated annealing and simplex. [3] Plot on the right shows the result for the gradient directed algorithm. While the older techniques take between 500 and 3500 iteration to converge because in the absence of the gradient information they need to perform extensive sampling of the control space, the gradient directed algorithm converges to the same solution in as little as 20 iterations. The use of the neural network emulator offers, therefore, a two orders of magnitude reduction in the number of iterations and a two orders of magnitude reduction in the execution time of each iteration.

6.3.1 Gradient Descent vs Momentum

Figures 6.13–6.15 compare the convergence of the simple gradient descent and the gradient descent with the momentum term on the problem of control synthesis. Section 3.4.2 describes in detail the advantage of using the momentum term during the update step. The two methods differ in the implementation of the update rule.

$$\mathbf{x}^{l+1} = \mathbf{x}^{l} + \delta\mathbf{x}^{l+1}. \tag{6.1}$$

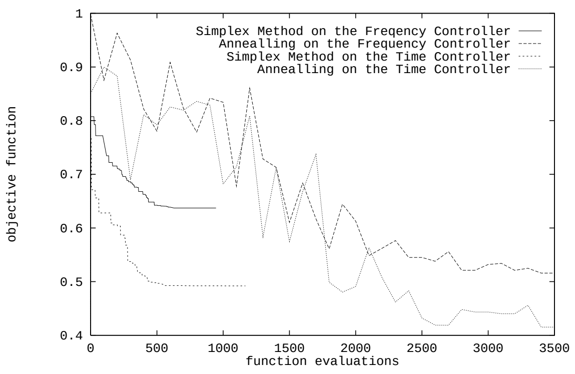
where the simple gradient descent defines $\delta\mathbf{w}^{l+1}$ as

$$\delta\mathbf{x}^{l+1} = -\eta_x \nabla_{\mathbf{x}} E^{\tau}(\mathbf{x}^{l}), \tag{6.2}$$

and the gradient descent with momentum defines this term as

$$\delta\mathbf{x}^{l+1} = -\eta_x \nabla_{\mathbf{x}} E^{\tau}(\mathbf{x}^{l}) + \alpha_x \delta\mathbf{x}^{l}. \tag{6.3}$$

[3]In Fig. 6.12, the plot on the left shows the results for the controller defined using both global and local basis functions. The plot on the right that illustrates the progress of learning of the gradient directed search shows the result for the controller defined using the local basis functions only.



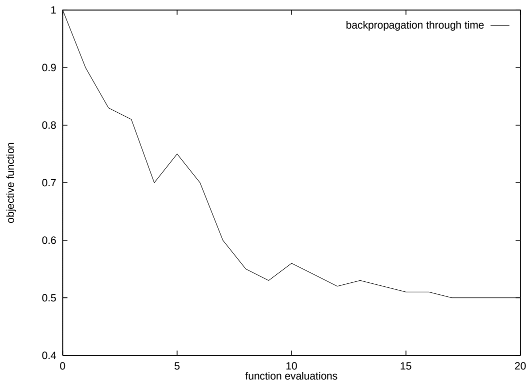
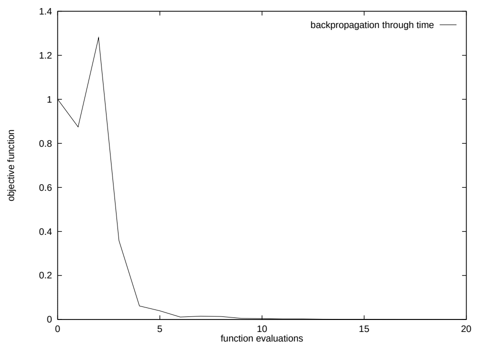


Figure 6.12: Convergence rate comparison between the directed and the undirected optimization methods on the problem of locomotion learning for the dolphin model. The plot on the left shows the progress of learning for two optimization methods that do not use the gradient information: simulated annealing and the simplex method. The plot on the right shows the convergence of the algorithm that uses gradient to direct the search. The technique that uses gradient converges in 20 iteration, while the other techniques take between 500 and 3500 iterations to find the solution.



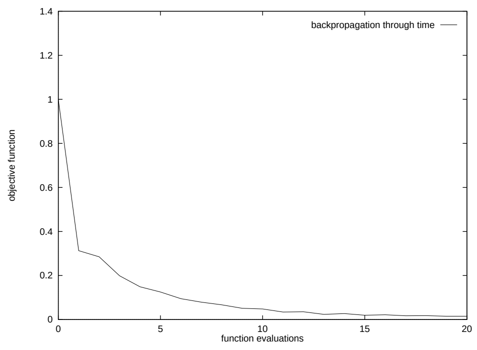


Figure 6.13: Progress of learning for the 3-link pendulum problem. The plot on the left was produced using the momentum term during, and therefore, converges faster than simple gradient descent shown on the right.

Fig. 6.13 illustrates the progress of learning for the 3-link pendulum control learning problem described in Section 6.2 and shown in Fig. 6.8. The results obtained using the momentum term are shown in the plot on the left and were generated using the following parameters: $\eta_x = 2.0$, $\alpha_x = 0.5$. The results obtained using the simple gradient descent are shown in the plot on the right and were generated using $\eta_x = 1.2$—the largest learning rate that would converge. Clearly, the momentum term decreases the error much more rapidly and permits a larger learning rate.

Fig. 6.14 illustrates the progress of learning for the lunar lander problem described in Section 6.2 and shown in Fig. 6.9. The results obtained using the momentum term are shown in the plot on the left and were generated using the following parameters: $\eta_x = 1.5$, $\alpha_x = 0.5$. The results obtained using the simple gradient descent are shown in the plot on the right and were generated using $\eta_x = 1.0$—the largest learning rate that would converge. As in the previous example, the momentum term decreases the error much more rapidly

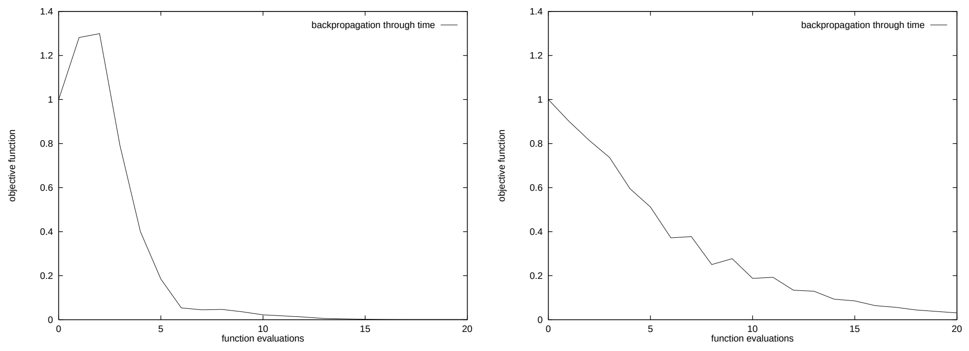


Figure 6.14: Progress of learning for the landing problem. The plots show the objective as a function of the iteration of the control learning algorithm. The plot on the left was produced using the momentum term, and therefore, converges faster than simple gradient descent shown on the right.

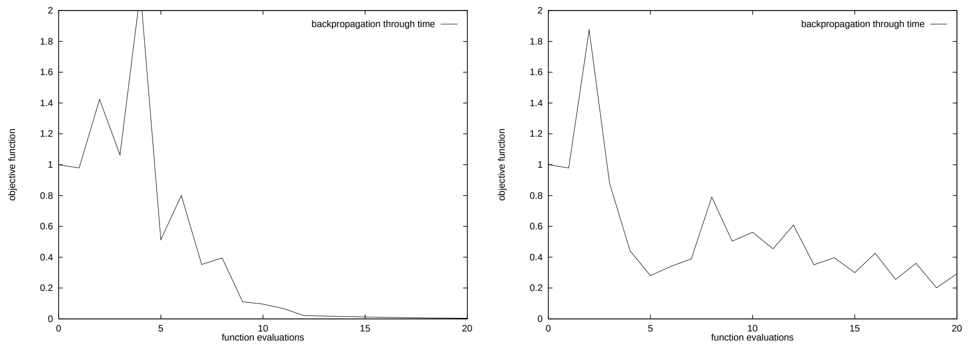


Figure 6.15: Progress of learning for the parking problem. The plots show the objective as a function of the iteration of the control learning algorithm. The plot on the left was produced using the momentum term, and therefore, converges faster than simple gradient descent shown on the right.

and enables a larger learning rate.

Fig. 6.15 illustrates the progress of learning for the parking problem described in Section 6.2 and shown in Fig. 6.10. The results obtained using the momentum term are shown in the plot on the left and were generated using the following parameters: $\eta_x = 1.5$, $\alpha_x = 0.5$. The results obtained using the simple gradient descent are shown in the plot on the right and were generated using $\eta_x = 1.5$. Although both strategies use the same learning rate, the momentum term damps the oscillation and therefore converges faster.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

Animation through the numerical simulation of physics-based graphics models offers unsurpassed realism, but it can be computationally demanding. Likewise, finding controllers that enable physics-based models to produce desired animations usually entails formidable computational cost. This thesis demonstrates the possibility of replacing the numerical simulation and control of model dynamics with a dramatically more efficient alternative. In particular, we propose the NeuroAnimator, a novel approach to creating physically realistic animation that exploits neural networks. NeuroAnimators are automatically trained off-line to emulate physical dynamics through the observation of physics-based models in action. Depending on the model, its neural network emulator can yield physically realistic animation one or two orders of magnitude faster than conventional numerical simulation. The efficiency is due to the low cost of the neural network evaluation and the fact that the network can accurately predict a long chain of evaluations of the physical simulator in a single step. Furthermore, by exploiting the network structure of the NeuroAnimator, we introduce a fast algorithm for learning controllers that enables either physics-based models or their neural network emulators to synthesize motions satisfying prescribed animation goals. We demonstrate NeuroAnimators for passive and active (actuated) rigid body, articulated, and deformable physics-based models.

To minimize the approximation error of the emulation, we construct a structured NeuroAnimator that has intrinsic knowledge about the underlying model built into it. The main subnetwork of the structured NeuroAnimator predicts the state changes of the model in its local coordinate system. A cascade of smaller subnetworks forms an interface between the main subnetwork and the inputs and outputs of the NeuroAnimator. To limit the approximation error that accumulates during the emulation of the deformable models, we introduce the regularization step that minimizes the deformation energy.

For complex dynamical systems with large state spaces we introduce hierarchical emulators that can be trained much more efficiently. The networks in the bottom level of this two level hierarchy are responsible for the emulation of different subsystems of the model, while the network at the top level of the hierarchy combines the results obtained by the lower level.

Except for very simple cases for which one can derive an analytical description, the physical systems used in computer graphics are not differentiable. Controller synthesis without gradient information necessitates brute force exploration of the control space that requires many costly physical simulations and is therefore painfully slow. However, the NeuroAnimator forms a differentiable model of the underlying physical system and, hence, forms a basis for an efficient control algorithm that uses the gradient to guide the search through the control space. In essence, the algorithm integrates the model over time to predict the future result and then differentiates the error in the outcome back in time to compute the effect of current actions on future results. This strategy proves very efficient since it replaces costly physical simulation with inexpensive neural emulation, and it synthesizes solutions to complex control problems in a few iterations by using the gradient.

The heavy computational burden of physical simulation and the lack of suitable control algorithms has retarded the penetration of physics-based models into commercial computer graphics systems. The connectionist approach developed in this thesis addresses both of these issues simultaneously in a unified fashion. It promises to help make physical realism in computer animation ubiquitous.

A neural network emulation offers a convenient handle for trading efficiency and realism. For example, if we care more for efficiency than for realism we can approximate the physical model using a network with a reduced number of weights and units, or we can train the network to make larger time steps then the physical simulator. Connectionist emulation therefore gives the user control over degree of physical realism.

7.2 Future Research Directions

7.2.1 Model Reconstruction

In the future, we foresee neural network emulators, similar to those presented in the dissertation, finding applications in the reconstruction of physically realistic models. Suppose, for example, that we have the training data describing physically correct state transitions of a model, but not its physical properties, nor the control forces that caused these transitions. The case we have in mind is that of motion capture which certainly produces physically realistic data, but excludes a lot of information about the underlying physical model such as control and contact forces. Given that the training data has enough diversity, we should be able to compose an emulator for the model underlying the captured data that recovers all its salient physical properties. In fact, we should be able to reconstruct the controllers responsible for the state transitions in the training data without much difficulty. We think this should be a very interesting line of research.

7.2.2 Emulator Improvements

We want to test the effect of using a committee of networks on the generalization quality of the network. This method trains many different candidate networks and then combines them together to form a committee (Perrone, 1993; Perrone and Cooper, 1994). This leads to significant improvements in generalization, but incurs a performance penalty due to the evaluation of multiple networks during each emulation step.

Yet another approach to improving network generalization we would like to evaluate is the use of adaptive mixtures of local experts to divide the learning into subtasks, each of which can be solved by a small and efficient expert network (Jacobs et al., 1991; Jordan and Jacobs, 1994).

7.2.3 Neural Controller Representation

In the future, we would like to replace the current controller representation that uses control variables to describe a set of control functions with a neural network. The existing controller can learn to perform only a single control task while the controller network should be able to learn a variety of them. In fact, it is possible that the controller network trained on a repertoire of simple basic control problems will likely generalize to similar yet different tasks. We hope that the connectionist controller offers a sufficiently elaborate representation to enable the emergence of complex behavior. We foresee that a controller network trained on a variety of simple, low-level tasks, such as swimming forward and turning, will automatically form an abstraction of controllers that solve higher-level problems, such as target tracking. Furthermore, we believe that the connectionist formulation offers uniformity that will make the interface between the separate control networks much more intelligible and elegant.

7.2.4 Hierarchical Emulation and Control

In the future, as we seek computer models that solve increasingly difficult tasks requiring extensive planning and taking a long time to execute, we will need to build emulators at different temporal scales. Even the best emulators cannot predict the behavior of the underlying model accurately forever. We foresee that the control for complex tasks will therefore have to be done hierarchically, where the control algorithm first solves the problem at a high level of abstraction described at a coarse temporal level, and then learns to perform each high-level task using low-level actions that take less time to execute.

7.2.5 NeuroAnimators in Artificial Life

In recent years, a new computation field has emerged, dubbed Artificial Life, that seeks to gain a better understanding of complex biological systems through their computer simulation. Biologically inspired models of animals, or animats, have also been of interest to computer animators who believe that such models will lead to more realistic results. The work presented here is of interest to both disciplines since it makes computer models more reactive and life-like through the use of biologically inspired neural networks.

One possible interpretation for the NeuroAnimators, that would be appealing from the Artificial Life point of view, is that the emulation represents the retrospective mode of the animat, while the simulation represents its active mode, and the animat uses them interchangeably to make better decisions about the future actions. In this scenario, the model is normally engaged in the execution mode, but breaks occasionally from it in order to reflect on its actions and plan future behavior. This is an interesting topic for future work.

Appendix A

Forward Pass Through the Network

The following is a C++ function for calculating the outputs of a neural network from the inputs. It implements the core loop that takes a single super timestep in an animation sequence with a trained NeuroAnimator.

```
BasicNet::forwardStep(void)
{
  int i,j,k;

  double *input = inputLayer.units;
  double *hidden = hiddenLayer.units;
  double *output = outputLayer.units;
  double **ww = inputHiddenWeights;
  double **vv = hiddenOutputWeights;

  // compute the activity of the hidden layer
  for (j=0;j<hiddenSize;j++) {
    hidden[j] = biasHiddenWeights[j];
    for (i=0;i<inputSize;i++)
      hidden[j] += input[i]*ww[i][j];
    hidden[j]=hiddenLayer.transFunc(hidden[j]);
  }

  // compute the activity of the output layer
  for (k=0;k<outputSize;k++) {
    output[k] = biasOutputWeights[k];
    for (j=0;j<hiddenSize;j++)
      output[k] += hidden[j]*vv[j][k];
    output[k]=outputLayer.transFunc(output[k]);
  }

}
```

Appendix B

Weight Update Rule

In Section 3.3.1 we have presented the weight update rule that uses the on-line gradient descent to train a network to approximate an example set $(\mathbf{x}^\tau, \mathbf{y}^\tau)$. Here we derive an efficient, recursive implementation of the algorithm that constitutes the essential part of the backpropagation algorithm (Rumelhart, Hinton and Williams, 1986). Although the derivation is for a specific case of a simple feedforward neural network with one hidden layer, we also show how to generalize the algorithm to a feedforward network with an arbitrary topology. Fig. B.1 introduces the notation.

Recall that the algorithm defines the network approximation error as

$$E^\tau(\mathbf{w}) = E(\mathbf{x}^\tau, \mathbf{w}) = ||\Phi(\mathbf{x}^\tau) - \mathbf{N}(\mathbf{x}^\tau, \mathbf{w})||, \tag{B.1}$$

and it seeks to minimize the objective

$$E(\mathbf{w}) = \frac{1}{2}\sum_{\tau=1}^{n} E^\tau(\mathbf{w}), \tag{B.2}$$

where n is the number of training examples. The on-line gradient descent implementation of the algorithm adjusts the network weights after each training example k:

$$\mathbf{w}^{l+1} = \mathbf{w}^l - \eta_w \nabla_{\mathbf{w}} E^\eta(\mathbf{w}^l) \tag{B.3}$$

where $\eta_w < 1$ denotes the *weight update learning rate*, and l specifies the iteration of the minimization step.

We derive here an efficient, recursive rule for computing the derivatives in (B.3). The algorithm first computes the derivative of the error with respect to the layer of weights directly adjacent to the outputs, and then uses this result recursively to compute the derivatives with respect to the earlier layers of weights. The name of the backpropagation algorithm stems from this recursive computation, that in essence propagates the derivative information from the network outputs back to its inputs.

We start the derivation by computing the error derivatives with respect to the hidden-to-output connections. Using the chain rule of differentiation we get the following expression

$$\frac{\partial E^\tau(\mathbf{w})}{\partial w_{jk}} = \frac{\partial E^\tau(\mathbf{w})}{\partial y_k}\frac{\partial y_k}{\partial w_{jk}} \tag{B.4}$$

Applying the chain rule twice, we get a similar expression for the derivatives with respect

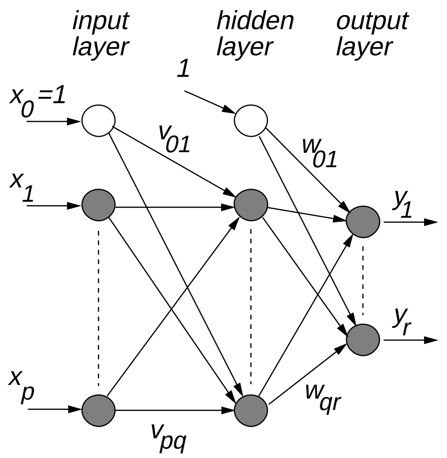


Figure B.1: Three-layer feedforward neural network **N**. Bias units are not shaded. x_i denotes the activation of input unit i, h_j—of hidden unit j, and y_k—of output unit k; v_{ij} describes the weight between input unit i and hidden unit j, and w_{jk} describes the weight between hidden unit j and output unit k.

to the input-to-hidden connections

$$\frac{\partial E^{\tau}(\mathbf{w})}{\partial v_{ij}} = \frac{\partial E^{\tau}(\mathbf{w})}{\partial h_j}\frac{\partial h_j}{\partial v_{ij}} = \frac{\partial h_j}{\partial v_{ij}}\sum_{k=1}^{r}\frac{\partial E^{\tau}(\mathbf{w})}{\partial y_k}\frac{\partial y_k}{\partial h_j} \tag{B.5}$$

All the derivatives occurring on the left side of (B.4) and (B.5) can now be easily calculated. Based on the notation introduced in Fig. B.1, we have the following expressions describing the activations of hidden and output units

$$h_j = g_h(\sum_{i=1}^{p} x_i v_{ij}) = g_h(\mathbf{x}\mathbf{v}_j) \tag{B.6}$$

$$y_k = g_y(\sum_{j=1}^{q} h_j w_{jk}) = g_y(\mathbf{h}\mathbf{w}_k) \tag{B.7}$$

where g_h and g_y are respectively the activation functions of the hidden and the output layer. We use these expressions to derive the partial derivatives used in (B.4) and (B.5)

$$\frac{\partial y_k}{\partial w_{jk}} = h_j g_y'(\mathbf{h}\mathbf{w}_k) \tag{B.8}$$

$$\frac{\partial y_k}{\partial h_j} = w_{jk} g_y'(\mathbf{h}\mathbf{w}_k) \tag{B.9}$$

$$\frac{\partial h_j}{\partial v_{ij}} = x_i g_h'(\mathbf{x}\mathbf{v}_j) \tag{B.10}$$

$$\frac{\partial h_j}{\partial x_i} = v_{ij} g_h'(\mathbf{x}\mathbf{v}_j) \tag{B.11}$$

where g_h' and g_y' indicate the derivatives of the activation functions. Assuming that the approximation error is defined as the squared distance between the output of the network and the desired output and from B.2, we can additionally write

$$\frac{\partial E^{\tau}(\mathbf{w})}{\partial y_k} = \frac{1}{2}\frac{\partial (t_k - y_k)^2}{\partial y_k} = -(t_k - y_k) = E_k^{\tau} \tag{B.12}$$

where t_k is the target output for unit k and y_k is the true output for this unit. Substituting (B.12) and (B.8) into (B.4) we get the expression for the error derivatives with respect to the hidden-to-output weights

$$\frac{\partial E^{\tau}(\mathbf{w})}{\partial w_{jk}} = E_k^{\tau} \cdot h_j \cdot g_y'(\mathbf{h}\mathbf{w}_k) = h_j \delta_k \tag{B.13}$$

where we have defined

$$\delta_k = E_k^{\tau} \cdot g_y'(\mathbf{h}\mathbf{w}_k) \tag{B.14}$$

Substituting (B.10), (B.12) and (B.9) into (B.5) we get the expression for the error derivatives with respect to the input-to-hidden weights

$$\frac{\partial E^{\tau}(\mathbf{w})}{\partial v_{ij}} = x_i \cdot g_h'(\mathbf{x}\mathbf{v}_j) \sum_{k=1}^{r} E_k^{\tau} \cdot w_{jk} \cdot g_y'(\mathbf{h}\mathbf{w}_k). \tag{B.15}$$

Using (B.14), we can rewrite this expression as

$$\frac{\partial E^{\tau}(\mathbf{w})}{\partial v_{ij}} = x_i \cdot g_h'(\mathbf{x}\mathbf{v}_j) \sum_{k=1}^{r} w_{jk}\delta_k = x_i \delta_j \tag{B.16}$$

where we have defined

$$\delta_j = g_h'(\mathbf{x}\mathbf{v}_j) \sum_{k=1}^{r} w_{jk}\delta_k \tag{B.17}$$

Based on (B.3), we can finally write the on-line gradient descent rule for the hidden-to-output layer

$$w_{jk}^{l+1} = w_{jk}^{l} - \eta_w h_j \delta_k, \tag{B.18}$$

and for the input-to-hidden layer

$$v_{ij}^{l+1} = v_{ij}^{l} - \eta_w x_i \delta_j. \tag{B.19}$$

Note that both update rules have the same form, and differ only in the definition of δ's. In fact, since (B.17) gives a recursive rule for computing δ's for the given layer of units in terms of δ's from the previous layer, update rule (B.19) can be used in a feedforward networks with an arbitrary number of layers. Incremental computation of the derivatives makes the backpropagation algorithm very efficiently.

Computation of the derivatives can be further simplified if we take the implementation specifics of an emulator under account. Since the emulator uses the identity function as the input activation function g_x and the output activation function g_y, the functions derivatives, g_x' and g_y', become the identity. Also, the emulator always uses the *logistic sigmoid* function as the activation function for the hidden layer, and this function has a useful property that its derivative can be expressed in terms of the function itself: $sig'(x) = sig(x)(1 - sig(x))$.

These facts enable us to write (B.17) as

$$\delta_j = sig'(\mathbf{x}\mathbf{v}_j)\sum_{k=1}^{r} w_{jk}\delta_k = h_j(1-h_j)\sum_{k=1}^{r} w_{jk}\delta_k, \tag{B.20}$$

and (B.14) as

$$\delta_k = E_k^{\tau} \cdot g_y'(\mathbf{h}\mathbf{w}_k) = E_k^{\tau}. \tag{B.21}$$

Since the new set of update rules reuses unit activations computed during the forward pass through the network, it makes the backpropagation step simpler and more efficient.

The following is a C++ function implementing the online version of the weight update rule. We assume that before executing this function the network has been presented with an input pattern for which there is a known target pattern, and that the activations of the hidden and the output layers were calculated for the network based on this input using the function presented in Appendix A. The update rule for the bias weights is the same as for the other weights.

```
void
BasicNet::backpropUpdateWeightsStep(void)
{
  int i, j, k; double value;

  double *input = inputLayer.units;
  double *hidden = hiddenLayer.units;
  double *output = outputLayer.units;
  double *hiddenDelta = hiddenDeltaLayer.units;
  double *outputDelta = outputDeltaLayer.units;
  double **v = inputHiddenWeights;
  double **w = hiddenOutputWeights;

  // compute the output layer deltas
  for(k=0;k<outputSize;k++) {
    outputDelta[k] = -(target[k] - output[k]);
    ouputDelta[k] *= outputLayer.transDeriv(output[k]);
  }

  // update the hidden-to-output weights
  for (j=0;j<hiddenSize;j++)
    for(k=0;k<outputSize;k++) {
       w[j][k] -= learningRate * hidden[j] * outputDelta[k];
       biasOutputWeights[k] -= learningRate * outputDelta[k];
    }

  // compute the hidden layer deltas
  for (j=0;j<hiddenSize;j++) {
    hiddenDelta[j] = 0.0;
    for(k=0;k<outputSize;k++)
       hiddenDelta[j] += w[j][k] * outputDelta[k];
  hiddenDelta[j] *= hiddenLayer.transDeriv(hidden[j]);
}

// update the input-to-hidden weights
for(i=0;i<inputSize;i++)
  for (j=0;j<hiddenSize;j++) {
     v[i][j] -= learningRate * input[i] * hiddenDelta[j];
     biasHiddenWeights[j] -= learningRate * hiddenDelta[j];
  }
```

Appendix C

Input Update Rule

In Section 3.3.2 we have presented the input update rule that uses the on-line gradient descent to alter the control inputs of the network in order to achieve the desired output. This algorithm forms an essential part of the controller synthesis process described in Chapter 5. The input update rule differs from the traditional version of the backpropagation algorithm (Rumelhart, Hinton and Williams, 1986) in that it adjusts the inputs while keeping the weights constant. However, the essential philosophy of the backpropagation algorithm stays intact. For this reason, there is a significant overlap between the weight update rule (Appendix B) and the rule described here. Both derivations share the same notation introduced in Fig. B.1.

Recall that the algorithm seek to minimize the objective defined as

$$E(\mathbf{x}) = ||\mathbf{N}_d - \mathbf{N}(\mathbf{x})||^2 \tag{C.1}$$

where $\mathbf{N}_d$ denotes the desired output of the network. We can rewrite E as

$$E(\mathbf{x}) = \frac{1}{2}\sum_{k=1}^{r}(t_k - y_k)^2 = \sum_{k=1}^{r} E_k \tag{C.2}$$

The on-line gradient descent implementation of the algorithm adjusts the network inputs using the following rule

$$\mathbf{x}^{l+1} = \mathbf{x}^l - \eta_x \nabla_{\mathbf{x}} E(\mathbf{x}^l) \tag{C.3}$$

where $\eta_x < 1$ denotes the *input update learning rate*, and l describes the iteration of the optimization step.

We derive here an efficient, recursive rule for computing the derivatives in (C.3). Applying the chain rule of differentiation twice to the equation we get

$$\frac{\partial E}{\partial x_i} = \sum_{k=1}^{r} \frac{\partial E_k}{\partial x_i} = \sum_{k=1}^{r} \frac{\partial E_k}{\partial y_k}\frac{\partial y_k}{\partial x_i} = \sum_{k=1}^{r} \frac{\partial E_k}{\partial y_k} \sum_{j=1}^{q} \frac{\partial y_k}{\partial h_j}\frac{\partial h_j}{\partial x_i} \tag{C.4}$$

We rearrange the terms to get

$$\frac{\partial E}{\partial x_i} = \sum_{j=1}^{q}\sum_{k=1}^{r} \frac{\partial E_k}{\partial y_k}\frac{\partial y_k}{\partial h_j}\frac{\partial h_j}{\partial x_i} = \sum_{j=1}^{q} \frac{\partial h_j}{\partial x_i} \sum_{k=1}^{r} \frac{\partial E_k}{\partial y_k}\frac{\partial y_k}{\partial h_j} \tag{C.5}$$

Note that the bias units, which receive constant input signals and have therefore zero gradient, have been excluded from the summation term. Using (B.9) and (B.11), we simplify the notation by introducing the following expressions

$$\begin{aligned}
\delta_k &= \frac{\partial E_k}{\partial y_k} = -(t_k - y_k) \\
\delta_j &= \sum_{k=1}^{r} \delta_k \frac{\partial y_k}{\partial h_j} = \sum_{k=1}^{r} \delta_k \cdot w_{jk} \cdot g'_y(\mathbf{hw}_k) \\
\delta_i &= \sum_{j=1}^{q} \delta_j \frac{\partial h_j}{\partial x_i} = \sum_{j=1}^{q} \delta_j \cdot v_{ij} \cdot g'_h(\mathbf{xv}_j)
\end{aligned}$$

Based on (C.3), we can now write a very concise representation of the on-line gradient descent rule for the ith input unit

$$\mathbf{x}_i^{l+1} = \mathbf{x}_i^l - \eta_x \delta_i \tag{C.6}$$

Since the rule for the input δ's is defined recursively in terms of the δ's in the previous layers, the input update rule (C.6) can be used to compute the derivatives in a feedforward network with an arbitrary number of layers.

We use the *logistic sigmoid* function as the activation function for the hidden layer and the identity function for both the input and the output layer. The logistic sigmoid function has the property that its derivative can be expressed in terms of the function itself: $sig'(x) = sig(x)(1 - sig(x))$. These facts enable us to simplify the expression for δs

$$\begin{aligned}
\delta_j &= \sum_{k=1}^{r} \delta_k \cdot w_{jk} \cdot g'_y(\mathbf{hw}_k) = \sum_{k=1}^{r} \delta_k \cdot w_{jk} \\
\delta_i &= \sum_{j=1}^{q} \delta_j \cdot v_{ij} \cdot g'_h(\mathbf{xv}_j) = \sum_{j=1}^{q} \delta_j \cdot v_{ij} \cdot h_j(1 - h_j)
\end{aligned}$$

This makes the backpropagation step easier to implement and more efficient, because we can evaluate the input adjustments using solely the unit activations computed during the forward pass through the network.

The following is a C++ function implementing the online version of the input update rule. We assume that before executing this function the network has been presented with an input pattern for which there is a known target pattern, and that the activations of the hidden and the output layers were calculated for the network based on this input using the function presented in Appendix A.

```
void
BasicNet::backpropInputsUpdateRule(void)
{
  int i, j, k; double value;

  double *hidden = hiddenLayer.units;
  double *output = outputLayer.units;
  double *inputDelta = inputDeltaLayer.units;
  double *hiddenDelta = hiddenDeltaLayer.units;
  double *outputDelta = outputDeltaLayer.units;
  // compute the deltas for the output layer ( output deltas
  // depend on the definition of the error function)
  setOutputDeltas();

  // compute the deltas for the hidden layer
  for (j=0;j<hiddenSize;j++) {
    hiddenDelta[j] = 0.0;
    for (k=0;k<outputSize;k++) {
      value = hiddenOutputWeights[j][k]*outputLayer.transDeriv(output[k]);
      value *= outputDelta[k];
      hiddenDelta[j] += value;
    }
  }

  // compute the deltas for the input layer
  for (i=0;i<inputSize;i++) {
    inputDelta[i] = 0.0;
    for (j=0;j<hiddenSize;j++) {
      value = inputHiddenWeights[i][j]*hiddenLayer.transDeriv(hidden[j]);
      value *= hiddenDelta[j];
      inputDelta[i] += value;
    }
  }

  // update the inputs to the network
  for (i=0;i<inputSize;i++)
    inputs[i] += learningRate*inputDelta[i];

}
```

Appendix D

Quaternions

A quaternion can represent an arbitrary 3D rotation using only four parameters, and rotation interpolation in the quaternion space is simple and smooth. These two properties make the quaternion encoding of rotation a better choice then its two alternatives: the Euler angles and the rotation matrix. The Euler angles, such as yaw, pitch, and roll, cannot represent an arbitrary rotation because of the singularity that occurs when the two rotations axes collapse into one. The rotation matrix, on the other hand, requires as many as nine parameters to encode the rotation and the interpolation of two rotation matrices cannot be performed directly.

For the reasons outlined above we choose to represent the joint orientations of a rigid-body using quaternions. However, the emulation and the control learning using quaternions requires a distinct set of operations. Specifically, to compute the change in the orientation of the model, we need to be able to subtract two quaternion rotations. We also need to know how to interpolate smoothly between two joint orientations represented as quaternions to obtain the motion of the model in-between the super timesteps. Finally, for the purposes of backpropagation through time we need to know how to differentiate operations that involve quaternions.

This appendix reviews the quaternion algebra related to our work. First we introduce the quaternion notation as presented in the paper by Hart *et al.* (Hart, Francis and Kaufmann, 1994). This includes a matrix representation for quaternion multiplication used by the emulation step. Subsequently, we review quaternion interpolation as presented in (Shoemake, 1985). Finally, we derive the quaternion multiplication differentiation step, and introduce the quaternion rotation error measure used to evaluate of the objective function.

D.1 Quaternion Representation

A quaternion $\mathbf{q} = r + x\mathbf{i} + y\mathbf{j} + z\mathbf{k} = r + \mathbf{v}$ consists of a real part r and a pure part $\mathbf{v} = x\mathbf{i} + y\mathbf{j} + z\mathbf{k}$. Vectors $\mathbf{i}$, $\mathbf{j}$, $\mathbf{k}$ signify the three-dimensional vectors

$$\begin{aligned} i &= (1,0,0), && \text{(D.1)} \\ j &= (0,1,0), && \text{(D.2)} \\ k &= (0,0,1). && \text{(D.3)} \end{aligned}$$

Let $\mathbf{q}_1 = r_1 + \mathbf{v}_1$ and $\mathbf{q}_2 = r_2 + \mathbf{v}_2$ be the two quaternions. Their sum is

$$\mathbf{q}_1 + \mathbf{q}_2 = (r_1 + r_2) + (\mathbf{v}_1 + \mathbf{v}_2), \tag{D.4}$$

and their product is

$$\mathbf{q}_1\mathbf{q}_2 = r_1 r_2 + \mathbf{v}_1 \cdot \mathbf{v}_2 + r_1\mathbf{v}_2 + r_2\mathbf{v}_1 + \mathbf{v}_1 \times \mathbf{v}_2. \tag{D.5}$$

The *magnitude* of q is defined as

$$\|\mathbf{q}\| = \sqrt{r^2 + \mathbf{v}\cdot\mathbf{v}}.$$

Let $\mathbf{q} = a + b\mathbf{u}$ be a quaternion such that $\|\mathbf{u}\| = 1$ is the imaginary unit three-vector. Its *conjugate* is $\bar{\mathbf{q}} = a - b\mathbf{u}$, and its magnitude is

$$\|\mathbf{q}\| = \mathbf{q}\bar{\mathbf{q}} = \sqrt{a^2 + b^2}. \tag{D.6}$$

Any quaternion $\mathbf{q} = a + \mathbf{v}$ can be represented as $\mathbf{q} = a + b\mathbf{u}$ through the relation

$$\mathbf{u} = \frac{x}{\|\mathbf{v}\|}\mathbf{i} + \frac{y}{\|\mathbf{v}\|}\mathbf{j} + \frac{z}{\|\mathbf{v}\|}\mathbf{k}. \tag{D.7}$$

D.2 Quaternion Rotation

Rotations are represented by *unit quaternions*, i.e., the quaternions of unit magnitude. Shoemake (Shoemake, 1985) represents a rotation of θ about the axis $\mathbf{u}$ as

$$\mathbf{q} = \cos\frac{1}{2}\theta + \sin\frac{1}{2}\theta\mathbf{u}. \tag{D.8}$$

Hart *et al.* employ the notation used in engineering sciences, that represents a complex number of unit magnitude as $e^{\mathbf{i}\theta} = \cos\theta + \mathbf{i}\sin\theta$, to produce a very compact quaternion representation

$$\mathbf{q} = e^{\frac{1}{2}\theta\mathbf{u}}. \tag{D.9}$$

D.2.1 Rotation Negation

From (D.6) it follows that to invert a unit quaternion, we simply negate its pure part

$$\mathbf{q}^{-1} = \bar{\mathbf{q}}. \tag{D.10}$$

D.2.2 Rotation Summation

We multiply two quaternions to sum the rotations that they represent, analogously to the operation done on the rotation matrices

$$e^{\frac{1}{2}\theta\mathbf{u}} = e^{\frac{1}{2}\theta_1\mathbf{u}_1} \cdot e^{\frac{1}{2}\theta_2\mathbf{u}_2} = \mathbf{q}_1\mathbf{q}_2 \tag{D.11}$$

where the multiplication is done according to the rule (D.5). Note, that it is much more efficient to add rotations represented as quaternions then it is when they are represented using rotation matrices.

D.3 Rotation Interpolation

Linear interpolation of two rotations, $\mathbf{q}_1$ and $\mathbf{q}_2$, requires that the rotation changes at a constant speed. Shoemake showed that this can be done by interpolating on the unit sphere, along the great circle arc between the two rotations. He gives the following formula for the *spherical linear interpolation* from $\mathbf{q}_1$ to $\mathbf{q}_2$, with parameter u moving from 0 to 1

$$slerp(\mathbf{q}_1, \mathbf{q}_2, u) = \frac{\sin(1-u)\theta}{\sin\theta}\mathbf{q}_1 + \frac{\sin u\theta}{\sin\theta}\mathbf{q}_2, \tag{D.12}$$

where $q_1 \cdot q_2 = \cos\theta$.

D.4 Matrix Representation of Quaternion Multiplication

The structured emulator introduced in Section 4.3 computes the joint orientation $\boldsymbol{\omega}_{t+\Delta t}$ by adding two quaternion rotations: $\boldsymbol{\omega}_t$ and $\Delta\boldsymbol{\omega}_t$ through quaternion multiplication according to the rule(D.5). However, for the purposes of a connectionist implementation it is far more convenient to represent quaternion multiplication in a matrix form that we present here.

Let $\mathbf{q}_1 = [\ q_1^r\ q_1^x\ q_1^y\ q_1^z\ ]$ and $\mathbf{q}_2 = [\ q_2^r\ q_2^x\ q_2^y\ q_2^z\ ]$ represent two quaternion rotations, and let $\mathbf{q} = [\ q^r\ q^x\ q^y\ q^z\ ]$ be the product of $\mathbf{q}_1$ and $\mathbf{q}_2$. Expanding the multiplication rule (D.5) we get the following set of equations

$$\begin{aligned} q^r &= q_1^r q_2^r - q_1^x q_2^x - q_1^y q_2^y - q_1^z q_2^z \\ q^x &= q_1^r q_2^x + q_1^x q_2^r + q_1^y q_2^z - q_1^z q_2^y \\ q^y &= q_1^r q_2^y + q_1^y q_2^r + q_1^z q_2^x - q_1^x q_2^z \\ q^z &= q_1^r q_2^z + q_1^z q_2^r + q_1^x q_2^y - q_1^y q_2^x \end{aligned} \tag{D.13}$$

which can be written concisely in a matrix form

$$\mathbf{q} = \mathbf{q}_1\mathbf{q}_2 = \mathbf{Q}_1\mathbf{q}_2 = \begin{bmatrix} q_1^r & -q_1^x & -q_1^y & -q_1^z \\ q_1^x & q_1^r & q_1^z & -q_1^y \\ q_1^y & -q_1^z & q_1^r & q_1^x \\ q_1^z & q_1^y & -q_1^x & q_1^r \end{bmatrix} \begin{bmatrix} q_2^r \\ q_2^x \\ q_2^y \\ q_2^z \end{bmatrix}. \tag{D.14}$$

D.5 Differentiation of Quaternion Rotation

Since the emulation step performs quaternion multiplication, the backpropagation control learning algorithm needs to compute the derivative of the output rotation $\mathbf{q}$ with respect to the input quaternions $\mathbf{q}_1$ and $\mathbf{q}_2$. We use the expanded version of the quaternion multiplication (D.13) to compute the Jacobian of $\mathbf{q}$ with respect to $\mathbf{q}_1$

$$\frac{\partial\mathbf{q}}{\partial\mathbf{q}_1} = \begin{bmatrix} q_2^r & -q_2^x & -q_2^y & -q_2^z \\ q_2^x & q_2^r & q_2^z & -q_2^y \\ q_2^y & -q_2^z & q_2^r & q_2^x \\ q_2^z & q_2^y & -q_2^x & q_2^r \end{bmatrix} \tag{D.15}$$

and with respect to $\mathbf{q}_2$

$$\frac{\partial \mathbf{q}}{\partial \mathbf{q}_2} = \begin{bmatrix} q_1^r & -q_1^x & -q_1^y & -q_1^z \\ q_1^x & q_1^r & -q_1^z & q_1^y \\ q_1^y & q_1^z & q_1^r & -q_1^x \\ q_1^z & -q_1^y & q_1^x & q_1^r \end{bmatrix}. \tag{D.16}$$

D.6 Orientation Error Metric in Quaternion Space

Since the control learning algorithm computes the error at the end of an emulation based on the state of the model, we need to be able to determine what this error is for the orientation defined as a quaternion. Let $\mathbf{q}_{M+1}$ be the orientation of the model at the end of the emulation, and let $\mathbf{q}_d = [\ q_d^r\ q_d^x\ q_d^y\ q_d^z\ ]$ be the desired orientation of the model at this point. We define the orientation error as

$$E_R = \frac{1}{2}(\mathbf{q}_u - \mathbf{q}_o)^2. \tag{D.17}$$

where $\mathbf{q}_u = [\ 1\ 0\ 0\ 0\ ]$ is the quaternion with zero rotation, and $\mathbf{q}_o = \mathbf{q}\bar{\mathbf{q}}_d$. Using the chain rule of differentiation, we obtain the following expression for the derivative of E_R with respect to $\mathbf{q}$

$$\frac{\partial E_R}{\partial \mathbf{q}} = \frac{\partial E_R}{\partial \mathbf{q}_o}\frac{\partial \mathbf{q}_o}{\partial \mathbf{q}} \tag{D.18}$$

where

$$\frac{\partial E_R}{\partial \mathbf{q}_o} = [q_o^r - q_u^r, q_o^x - q_u^x, q_o^y - q_u^y, q_o^z - q_u^z] \tag{D.19}$$

and

$$\frac{\partial \mathbf{q}_o}{\partial \mathbf{q}} = \begin{bmatrix} \bar{q}_d^r & -\bar{q}_d^x & -\bar{q}_d^y & -\bar{q}_d^z \\ \bar{q}_d^x & \bar{q}_d^r & \bar{q}_d^z & -\bar{q}_d^y \\ \bar{q}_d^y & -\bar{q}_d^z & \bar{q}_d^r & \bar{q}_d^x \\ \bar{q}_d^z & \bar{q}_d^y & -\bar{q}_d^x & \bar{q}_d^r \end{bmatrix}. \tag{D.20}$$

Appendix E

Description of the Models

This appendix describes the physical models used as the emulator prototypes. For the rigid bodies we include the SD/FAST script used to build the model and the forces computation function used by the physical simulator.

E.1 Multi-Link Pendulum

The following is an SD/FAST script used to generate the equations of motion for the three-link pendulum shown in Fig. 6.1. The model has 3 links—*link0* attaches to the ground, *link1* attaches to *link0*, and finally *link2* attaches to *link1*. The links connect through the pin joints that rotate around the y-axis. There is a gravity force acting on the model.

E.1.1 SD/FAST Script

```
gravity = 0 0 -9.8

body = link0  inb = ground  joint = pin
  mass = 1                   inertia = 5 0 0 0 5 0 0 0 1
  bodytojoint = 0 0 0.25     inbtojoint = 0 0 0
  pin = 0 1 0

body = link1  inb = link0    joint = pin
  mass = 1                   inertia = 5 0 0 0 5 0 0 0 1
  bodytojoint = 0 0 0.25     inbtojoint = 0 0 -0.25
  pin = 0 1 0

body = link2  inb = link1  joint = pin
  mass = 1                   inertia = 5 0 0 0 5 0 0 0 1
  bodytojoint = 0 0 0.25     inbtojoint = 0 0 -0.25
  pin = 0 1 0
```

E.1.2 Force Computation Function

The following is a C function for calculating the external torques acting on the joints of the model. Each torque is a sum of the control force and the friction force that depends on the angular velocity of the joint.

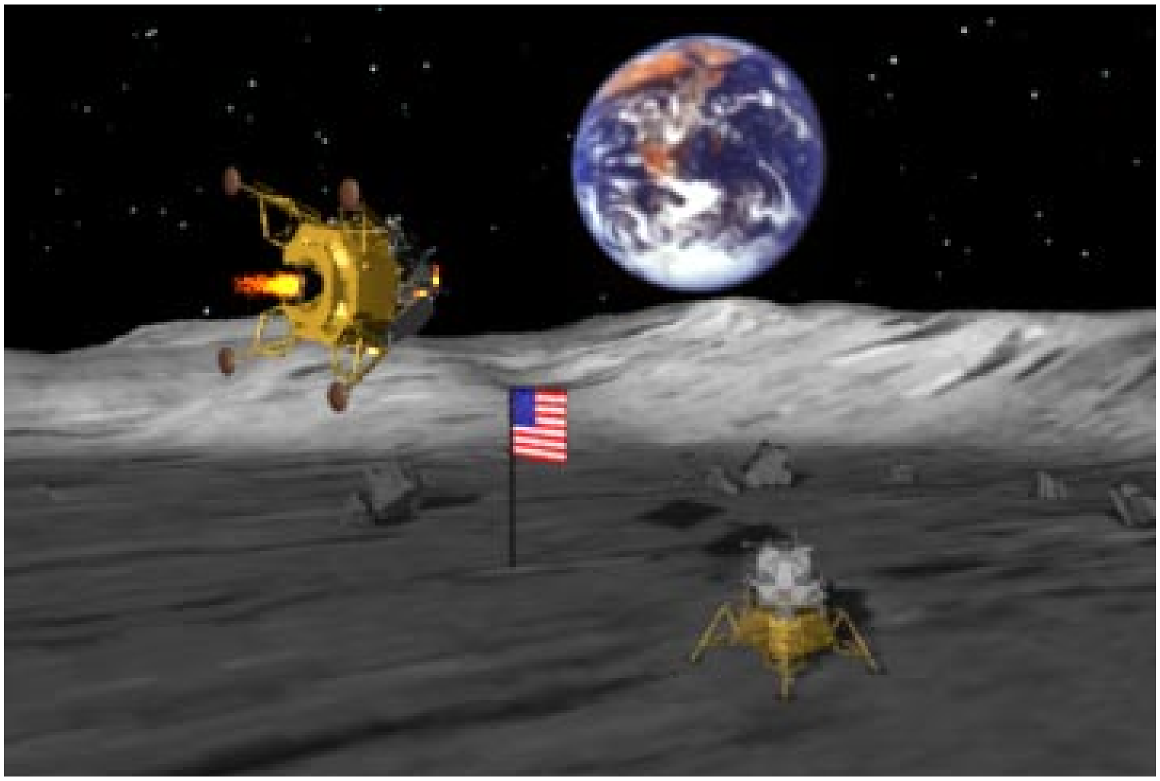


Figure E.1: The geometrical model of the lunar lander. Underlying it is a physical model that can rotate freely in 3D.

```
void sduforce(double time, double *pos, double *vel)
{
  int i; double torque;

  // loop through all the joints
  for (i=0;i<jointCount;i++) {
    // compute the torque
    torque = controlForce[i] - frictionCoef*vel[i];
    // apply the torque to the joint
    sdhinget(i,0,torque);
  }

}
```

E.2 Lunar Lander

The following is an SD/FAST script used to generate the equations of motion for the lunar lander model shown in Fig. E.1. Physical representation of the model consists of a rigid block that can rotate freely in 3D. The model has 4 internal controls—the main jet that propels the lander, and 3 orthogonal thrusters that change the orientation of the model.

E.2.1 SD/FAST Script

```
body = ship inb = ground joint = sixdof
  mass = 100.0
  inertia = 5 0 0 0 5 0 0 0 2
  bodytojoint = 0 -1 0.0 inbtojoint = 0 0 0
  pin = 1 0 0
  pin = 0 1 0
  pin = 0 0 1
```

E.2.2 Force Computation Function

```
extern int globalControlCount;
extern double* globalControls;

The following is a C function for calculating the external torques
acting on the joints of the model. Each torque is a sum of the control
force and the friction force that depends on the angular velocity of
the joint.

void sduforce(double time, double *pos, double *vel)
{
  // jet positions
  double shipPos[3] = { 0.0, 0.0, 0.0};
  double jet0[3] = { 0.25, 0.0, 0.0};
  double jet1[3] = {-0.25, 0.0, 0.0};


  // jet forces
  double mainJet[3] = {0.0,0.0,10.0};
  double azim0[3] = { 0.0, 0.5, 0.0};
  double azim1[3] = { 0.0,-0.5, 0.0};
  double incl0[3] = { 0.0, 0.0, 0.5};
  double incl1[3] = { 0.0, 0.0,-0.5};

  mainJet[2] *= globalControls[0];
  azim0[1] *= globalControls[1];
  azim1[1] *= globalControls[1];
  incl0[2] *= globalControls[2];
  incl1[2] *= globalControls[2];

  // apply the main jet force
  sdpointf(0,shipPos,mainJet);

  // apply the orientation jets
  sdpointf(0,jet0,azim0);
  sdpointf(0,jet0,incl0);
  sdpointf(0,jet1,azim1);
  sdpointf(0,jet1,incl1);

  // apply linear friction force
  double linIn[3], linOut[3];
  for (int i=0;i<3;i++) {
    linIn[i] = -0.2*vel[i];
  }
  sdtrans(-1,linIn,0,linOut);
  sdpointf(0,shipPos,linOut);

  // apply angular friction force
  double angIn[3], angOut[3];
  for (i=0;i<3;i++) {
    angIn[i] = -0.2*vel[i+3];
  }
  sdbodyt(0,angIn);
}
```

E.3 Truck

The following is an SD/FAST script used to generate the equations of motion for the truck model shown in Fig. E.2. Physical representation of the model consists of a rigid block that can slide on the plane $z = 0$, and rotate around the z axis. The model can exert two types of forces—one that accelerates/decelerates in the forward direction, and one that changes the orientation of the model.

E.3.1 SD/FAST Script

```
body = block inb = ground joint = planar
  mass = 100.0
  inertia = 5 0 0 0 5 0 0 0 2
  bodytojoint = 0 -1 0 inbtojoint = 0 0 0.25
  pin = 1 0 0
  pin = 0 1 0
  pin = 0 0 1
```

E.3.2 Force Computation Function

```
extern int globalControlCount;
extern double* globalControls;

void sduforce(double time, double *pos, double *vel)
{
  // change the orientation
  double stearPos[3] = { 0.0, 2.0, 0.0};
  // stearing forces
  double stear[3] = { 2.0, 0.0, 0.0};
```

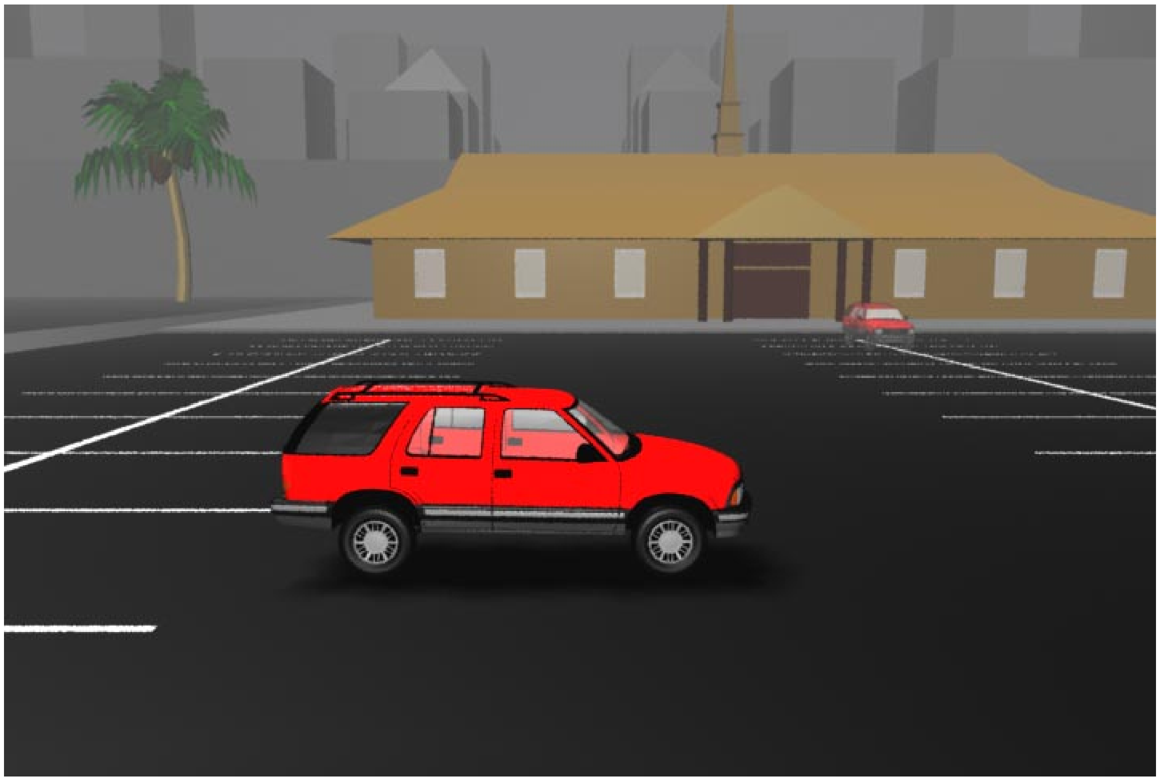


Figure E.2: The geometrical model of the truck. Underlying it is a physical model that consists of a rigid block that can slide on the plane $z = 0$, and rotate around the z axis.

```
  stear[0] *= globalControls[0];
  // apply the acceleration force
  sdpointf(0,stearPos,stear);

  Vector <double> forwardVec(0,1,0);
  Vector <double> sideVec(1,0,0);
  // get the car orientation
  double dircos[3][3];
  sdorient(0,dircos);
  Matrix3 orientMat(dircos);
  // convert the vectors to WCS
  forwardVec = orientMat*forwardVec;
  sideVec = orientMat*sideVec;

  // find the joint velocity in WCS
  double jointPos[3]; double absVel[3];
  sdgetbtj(0,jointPos);
  sdvel(0,jointPos,absVel);
  Vector <double> absVelVec; absVelVec.setVec(absVel);

  // compute the gas acceleration
  double gas[3] = { 0.0, 10.0, 0.0};
  gas[1] *= globalControls[1];
  // apply the aceleration force
  sdpointf(0,jointPos,gas);

  double force[3], forceOut[3];
  // compute the sideways friction
  double sideFactor = -200.0*sideVec.dot(absVelVec);
  Vector <double> sideForce(sideVec);
  sideForce *= sideFactor;
  sideForce.getVec(force);
  sdtrans(-1,force,0,forceOut);
  sdpointf(0,jointPos,forceOut);

  // compute the forward friction
  double forwardFactor = -0.2*forwardVec.dot(absVelVec);
  Vector <double> forwardForce(forwardVec);
  forwardForce *= forwardFactor;
  forwardForce.getVec(force);
  sdtrans(-1,force,0,forceOut);
  sdpointf(0,jointPos,forceOut);
}
```

E.4 Dolphin

E.4.1 Spring-Mass Model

The biomechanical model of the dolphin is constructed of nodal masses and springs, as is detailed in Fig. E.3. The model's dynamics is specified by the Lagrangian equations of motion

$$m_i \ddot{\mathbf{x}}_i + \gamma_i \dot{\mathbf{x}}_i + \sum_{j \in N_i} \mathbf{f}_{ij}^s = \mathbf{f}_i \tag{E.1}$$

where node i has mass m_i, position $\mathbf{x}_i(t) = [x_i(t), y_i(t), z_i(t)]$, velocity $\dot{\mathbf{x}}$, and damping factor γ_i, and where $\mathbf{f}_i$ is an external force. Spring S_{ij}, which connects node i to neighboring nodes $j \in N_i$, exerts the force $\mathbf{f}_{ij}^s(t) = -(c_{ij}e_{ij} + \gamma_{ij}\dot{e}_{ij})\mathbf{r}_{ij}/||\mathbf{r}_{ij}||$ on node i (and it exerts the force $-\mathbf{f}_{ij}^s$ on node j), where c_{ij} is the elastic constant, γ_{ij} is the damping constant, and $e_{ij}(t) = ||\mathbf{r}_{ij}|| - l_{ij}$ is the deformation of the spring with separation vector $\mathbf{r}_{ij}(t) = \mathbf{x}_j - \mathbf{x}_i$. The natural length of the spring is l_{ij}.

Some of the springs in the biomechanical model play the role of contractile *muscles.* Muscles contract as their natural length l_{ij} decreases under the autonomous control of the motor center of the artificial animal's brain (Tu and Terzopoulos, 1994). To dynamically contract a muscle, the brain must supply an *activation function* $a(t)$ to the muscle. This continuous time function has range $[0, 1]$, with 0 corresponding to a fully relaxed muscle of length l_{ij}^r and 1 to a fully contracted muscle of length l_{ij}^c. More specifically, for a muscle spring, $l_{ij} = al_{ij}^c + (1-a)l_{ij}^r$.

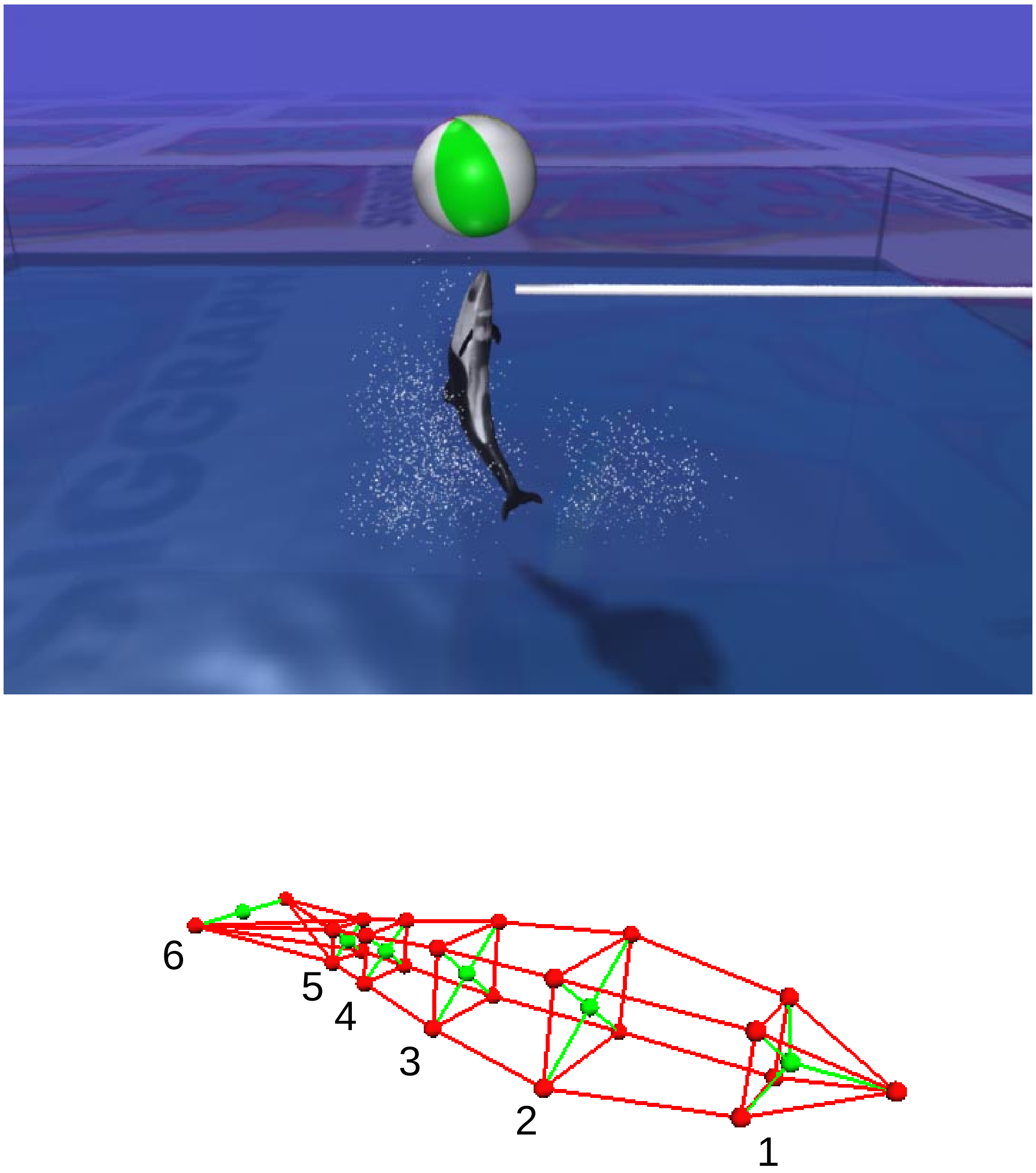


Figure E.3: The geometrical model of the dolphin (top) and the underlying biomechanical model (bottom). The model has six actuators consisting of a pair of muscles that share the same activation function. The numbers along the body indicate the local centers of mass used for building the hierarchical emulator. The cross-springs that maintain the structural integrity of the body have not been indicated.

Appendix F

Example Xerion Script

Section 4.3 proposed a special NeuroAnimator structure that minimizes the emulation approximation error. Fig. 4.4 shows the architecture of the structured NeuroAnimator that transforms the inputs to $\mathbf{N}_\Phi$ using a sequence of predefined transformations before feeding the signal through the specialized network $\mathbf{N}_\Phi^\sigma$. The following is a Xerion script that specifies and trains the network $\mathbf{N}_\Phi^\sigma$ used to build the NeuroAnimator for the lunar lander model described in Section E.2 and shown in Fig. E.1.

```
#! /u/xerion/uts/bin/bp_sh

# The network has 13 inputs, 50 hidden units, and
# 13 outputs. The hidden layer uses the logistic
# sigmoid as the activation function (default).
uts_simpleNet landerNet 13 50 13
bp_groupType landerNet.Hidden {HIDDEN DPROD LOGISTIC}

# Initialize the example set. Read
# the training data from a file.
set trainSet "landerNet.data"
uts_exampleSet   $trainSet
uts_loadExamples $trainSet landerNet.data

# Randomize the weights in the network.
random seed 3
uts_randomizeNet landerNet

# Initialize the minimizer and tell it to use
# the network and the training set defined above.
bp_netMinimizer mz
mz configure -net landerNet -exampleSet trainSet

# Start the training and save the weights
# of the network after the training is finished.
mz run
uts_saveWeights landerNet landerNet.weights
```

```
#---End of script---
```

F.1 Training Data

This section describes the conversion process that produces training data in the format required by the structured emulator. The example uses the physical model of the lunar lander described in Section E.2. Its state consists of four groups $\mathbf{s} = [\mathbf{p}\ \mathbf{q}\ \mathbf{v}\ \boldsymbol{\omega}]$ where $\mathbf{p} = [p^x\ p^y\ p^z]$ describes the position, $\mathbf{q} = [q^r\ q^x\ q^y\ q^z]$ describes the orientation, $\mathbf{v} = [v^x\ v^y\ v^z]$ describes the linear velocity, and $\boldsymbol{\omega} = [\omega^x\ \omega^y\ \omega^z]$ describes the angular velocity. The model has 4 internal actuators represented as $\mathbf{u} = [u^1\ u^2\ u^3\ u^4]$. The training data collected during the physical simulation of the model is represented in the format suitable for the input/output structure of the NeuroAnimator $\mathbf{N}_\Phi$:

$$\begin{array}{cc}
\{\mathbf{p}\,\mathbf{q}\,\mathbf{v}\,\boldsymbol{\omega}\,\mathbf{u}\}_{t_1} & \{\mathbf{p}\,\mathbf{q}\,\mathbf{v}\,\boldsymbol{\omega}\,\mathbf{u}\}_{t_1+\Delta t} \\
\{\mathbf{p}\,\mathbf{q}\,\mathbf{v}\,\boldsymbol{\omega}\,\mathbf{u}\}_{t_2} & \{\mathbf{p}\,\mathbf{q}\,\mathbf{v}\,\boldsymbol{\omega}\,\mathbf{u}\}_{t_2+\Delta t} \\
\vdots & \\
\{\mathbf{p}\,\mathbf{q}\,\mathbf{v}\,\boldsymbol{\omega}\,\mathbf{u}\}_{t_n} & \{\mathbf{p}\,\mathbf{q}\,\mathbf{v}\,\boldsymbol{\omega}\,\mathbf{u}\}_{t_n+\Delta t}
\end{array}$$

However, to train the structured emulator described in Section 4.3, the training data need to be transformed to a different format, suitable for the input/output structure of the subnet $\mathbf{N}^\sigma_\Phi$. Section 4.5.2 describes the data conversion process, and Fig. 4.13 illustrates the idea.

As a result of the conversion, the original set of inputs $\mathbf{x}$ undergoes the following transformation

$$\mathbf{x}^\sigma = \mathbf{T}^\sigma_x \mathbf{T}'_x \mathbf{x} \tag{F.1}$$

while the original set of outputs $\mathbf{y}$ is transformed through

$$\mathbf{y}^\sigma = (\mathbf{T}^\sigma_y \mathbf{T}'_y \mathbf{T}^\Delta_y)^{-1}\mathbf{y}. \tag{F.2}$$

The reader should note that the transformations applied to the input groups $\mathbf{p}_t$ and $\mathbf{q}_t$ produce zero vectors as output. Based on the facts that $= \mathbf{T}'_x\mathbf{p}_t = \mathbf{R}^T_t(\mathbf{p}_t - \mathbf{c}_t)$ and $\mathbf{p}_t = \mathbf{c}_t$, it follows that $\mathbf{T}'_x\mathbf{p}_t = [0\ 0\ 0]$. Similarly, from the facts that $\mathbf{T}'_x\mathbf{q}_t = \mathbf{Q}^T_t\mathbf{q}_t$ and $\mathbf{Q}^T_t\mathbf{q}_t = \bar{\mathbf{q}}_t\mathbf{q} = [1\ 0\ 0\ 0]$, it follows that $\mathbf{T}'_x\mathbf{q}_t = [1\ 0\ 0\ 0]$. Since there is no need to supply constant values as inputs to the network, the final form of the transformed training data looks as follows:

$$\begin{array}{cc}
\{\mathbf{v}^\sigma\,\boldsymbol{\omega}^\sigma\,\mathbf{u}^\sigma\}_{t_1} & \{\mathbf{p}^\sigma\,\mathbf{q}^\sigma\,\mathbf{v}^\sigma\,\boldsymbol{\omega}^\sigma\,\mathbf{u}^\sigma\}_{t_1+\Delta t} \\
\{\mathbf{v}^\sigma\,\boldsymbol{\omega}^\sigma\,\mathbf{u}^\sigma\}_{t_2} & \{\mathbf{p}^\sigma\,\mathbf{q}^\sigma\,\mathbf{v}^\sigma\,\boldsymbol{\omega}^\sigma\,\mathbf{u}^\sigma\}_{t_2+\Delta t} \\
\vdots & \\
\{\mathbf{v}^\sigma\,\boldsymbol{\omega}^\sigma\,\mathbf{u}^\sigma\}_{t_n} & \{\mathbf{p}^\sigma\,\mathbf{q}^\sigma\,\mathbf{v}^\sigma\,\boldsymbol{\omega}^\sigma\,\mathbf{u}^\sigma\}_{t_n+\Delta t}
\end{array}$$

Appendix G

Controller Representation

Discrete-time neural network emulation (4.1) requires discretization of the controller $\mathbf{u}(t)$. We parameterize the controller through discretization using basis functions. Mathematically, we express the ith control function as a B-spline

$$u_i(t) = \sum_{j=1}^{M} u_i^j B^j(t), \tag{G.1}$$

where the u_i^j are scalar parameters and the $B^j(t)$, $1 \leq j \leq M$ are (vector-valued) temporal basis functions. Backpropagation through time, used for control learning, constraints us to use local controller discretization, for which the parameters u_i^j are nodal variables and the $B^j(t)$ can be spline basis functions.[1] In a typical discretization the control variables do not span more than one super timestep Δt, which is often equivalent to 50 simulation time steps δt.

We use the Catmull-Rom family of splines to generate control functions that smoothly interpolate the control points (Catmull and Clark, 1974). We choose a member of this family for which the tangent vector at point u_i^j is parallel to the line connecting points u_i^{j-1} and u_i^{j+1}, as shown in Figure G. The representation for the jth segment of the ith control function is

$$u_i^j(t) = \frac{1}{2} \cdot \begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 2 & -5 & 4 & -1 \\ -1 & 0 & 1 & 0 \\ 0 & 2 & 0 & 0 \end{bmatrix} \begin{bmatrix} u_i^{j-1} \\ u_i^j \\ u_i^{j+1} \\ u_i^{j+2} \end{bmatrix}.$$

For the first segment of the curve $u_i^1(t)$ we make $u_i^0 = u_i^1$, i.e., we duplicate the first control point of each control function. Similarly, for the last segment of the curve $u_i^{M-1}(t)$ we make $u_i^{M+1} = u_i^M$, i.e., we duplicate the last control point of each control function.

[1]In our earlier work (Grzeszczuk and Terzopoulos, 1995), we proposed global controller discretization, for which the support of the $B^j(t)$ covers the entire temporal domain $t_0 \leq t \leq t_1$. We abandon this representation in this work since globally discretized controllers are not suitable for the current implementation of the control learning algorithm.

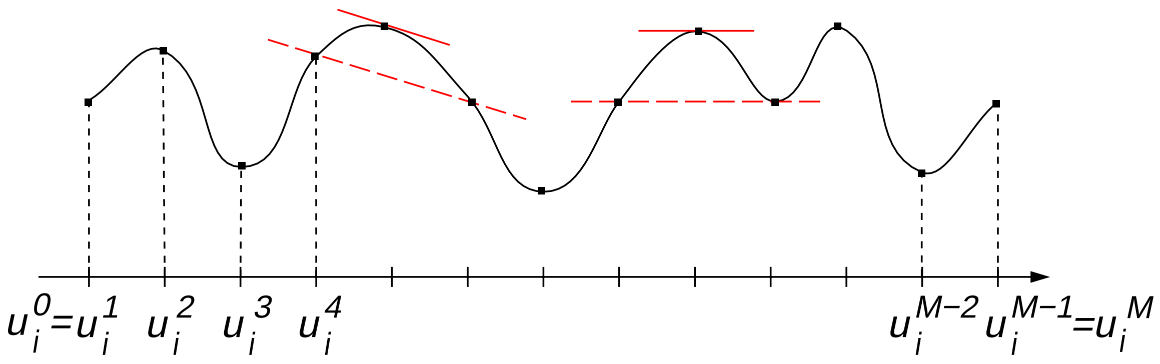


Figure G.1: A Catmull-Rom spline. The points are interpolated by the spline. Indicated by the straight line is the direction of the curve at each point. The direction is parallel to the line connecting two adjacent points. To extend the spline to the boundary points we duplicate the first and last points.

References

Albus, J. S. (1975). A new approach to manipulator control: The cerebellar model articulation controller (cmac). *Transactions of the ASME Journal of Dynamic Systems, Measurements, and Control*, 97:220–227.

Anderson, C. W. (1987). Strategy learning with multilayer connectionist representations. In *Proceedings of the Fourth International Workshop on Machine Learning*, pages 103–114.

Armstrong, W. W. and Green, M. (1985). The dynamics of articulated rigid bodies for purposes of animation. In Wein, M. and Kidd, E. M., editors, *Graphics Interface '85 Proceedings*, pages 407–415. Canadian Inf. Process. Soc.

Atkeson, C. G. and Reinkensmeyer, D. J. (1988). Using associative content-addressable memories to control robots. In *Proceedings of the IEEE Conference on Decision and Control*, pages 792–797, Austin, Tx.

Badler, N. I., Barsky, B. A., and Zeltzer, D., editors (1991). *Animation from instructions*, pages 51–93. Morgan Kaufmann.

Baraff, D. (1989). Analytical methods for dynamic simulation of non-penetrating rigid bodies. In Lane, J., editor, *Computer Graphics (SIGGRAPH '89 Proceedings)*, volume 23, pages 223–232.

Barto, A. G. (1990). Connectionist learning for control: An overview. In Miller, W. T., Sutton, R. S., and Werbos, P. J., editors, *Neural Networks for Control*, pages 5–58. The MIT Press.

Barto, A. G. and Jordan, M. I. (1987). Gradient following without back-propagation in layered networks. In Caudill, M. and Butler, C., editors, *Proceedings of the IEEE First Annual Conference on Neural Networks*, pages II629–II636, San Diego, CA. SOS Printing.

Barto, A. G., Sutton, R. S., and Anderson, C. W. (1983). Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man, and Cybernetics*, 13(5):834–846.

Barzel, R. and Barr, A. H. (1988). A modeling system based on dynamic constraints. In Dill, J., editor, *Computer Graphics (SIGGRAPH '88 Proceedings)*, volume 22, pages 179–188.

Bishop, C. M. (1995). *Neural Networks for Pattern Recognition.* Clarendon Press.

Brooks, R. (1991). Intelligence without representation. *Artificial Intelligence*, 47:139–159.

Brotman, L. S. and Netravali, A. N. (1988). Motion interpolation by optimal control. In Dill, J., editor, *Computer Graphics (SIGGRAPH '88 Proceedings)*, volume 22, pages 309–315.

Bruderlin, A. and Williams, L. (1995). Motion signal processing. In Cook, R., editor, *SIGGRAPH 95 Conference Proceedings*, Annual Conference Series, pages 97–104. ACM SIGGRAPH, Addison Wesley. held in Los Angeles, California, 06-11 August 1995.

Bryson, A. E. and Ho, Y.-C. (1969). *Applied Optimal Control.* Blaisdell, New York.

Carignan, M., Yang, Y., Thalmann, N. M., and Thalmann, D. (1992). Dressing animated synthetic actors with complex deformable clothes. In Catmull, E. E., editor, *Computer Graphics (SIGGRAPH '92 Proceedings)*, volume 26, pages 99–104.

Catmull, E. E. and Clark, J. H. (1974). A class of local interpolating splines. In Barnhill, R. E. and Riesenfeld, R. F., editors, *Computer Aided Geometric Design*, pages 317–326. Academic Press, New York.

Cohen, M. F. (1992). Interactive spacetime control for animation. In Catmull, E. E., editor, *Computer Graphics (SIGGRAPH '92 Proceedings)*, volume 26, pages 293–302.

Cybenko, G. (1989). Approximation by superposition of sigmoidal function. *Mathematics of Control Signals and Systems*, 2(4):303–314.

Dennis, J. E. and Schnabel, R. B. (1983). *Numerical Methods for Unconstrained Optimization and Nonlinear Equations.* Prentice-Hall, Englewood Cliffs, NJ.

Desbrun, M. and Gascuel, M. (1995). Animating soft substances with implicit surfaces. In Cook, R., editor, *SIGGRAPH 95 Conference Proceedings*, Annual Conference Series, pages 287–290. ACM SIGGRAPH, Addison Wesley. held in Los Angeles, California, 06-11 August 1995.

Duda, R. O. and Hart, P. E. (1973). *Pattern Classification and Scene Analysis.* Wiley, New York.

Fletcher, R. (1987). *Practical Methods for Optimization.* John Wiley, New York, second edition.

Foley, J. D., van Dam, A., Feiner, S. K., and Hughes, J. F. (1990). *Computer Graphics, Principles and Practice, Second Edition.* Addison-Wesley, Reading, Massachusetts. Overview of research to date.

Foster, N. and Metaxas, D. (1997). Modeling the motion of a hot, turbulent gas. In Whitted, T., editor, *SIGGRAPH 97 Conference Proceedings*, Annual Conference Series, pages 181–188. ACM SIGGRAPH, Addison Wesley. ISBN 0-89791-896-7.

Gill, P. E., Murray, W., and Wright, M. H. (1981). *Practical Optimization.* Academic Press, London.

Goh, C. J. and Teo, K. L. (1988). Control parameterization: A unified approach to optimal control problems with general constraints. *Automatica*, 24:3–18.

Grossberg, S. and Kuperstein, M. (1986). *Neural dynamic of adaptive sensory-motor control: Ballistic eye movements.* Elsevier, Amsterdam.

Grzeszczuk, R. and Terzopoulos, D. (1995). Automated learning of Muscle-Actuated locomotion through control abstraction. In Cook, R., editor, *SIGGRAPH 95 Conference Proceedings*, Annual Conference Series, pages 63–70. ACM SIGGRAPH, Addison Wesley. held in Los Angeles, California, 06-11 August 1995.

Guenter, B., Rose, C. F., Bodenheimer, B., and Cohen, M. F. (1996). Efficient generation of motion transitions using spacetime constraints. In Rushmeier, H., editor, *SIGGRAPH 96 Conference Proceedings*, Annual Conference Series, pages 147–154. ACM SIGGRAPH, Addison Wesley. held in New Orleans, Louisiana, 04-09 August 1996.

Hahn, J. K. (1988). Realistic animation of rigid bodies. In Dill, J., editor, *Computer Graphics (SIGGRAPH '88 Proceedings)*, volume 22, pages 299–308.

Hart, J. C., Francis, G. K., and Kaufmann, L. H. (1994). Visualizing quaternion rotation. *ACM Transactions on Graphics*, 13(3):256–276.

Hertz, J., Krogh, A., and Palmer, R. G. (1991). *Introduction to the Theory of Neural Computation.* Addison-Wesley.

Hinton, G. E. (1989). Connectionist learning procedures. *Artificial Intelligence*, 40:185–234.

Hodgins, J. K., Wooten, W. L., Brogan, D. C., and O'Brien, J. F. (1995). Animating human athletics. In Cook, R., editor, *SIGGRAPH 95 Conference Proceedings*, Annual Conference Series, pages 71–78. ACM SIGGRAPH, Addison Wesley. held in Los Angeles, California, 06-11 August 1995.

Hornik, K., Stinchcomb, M., and White, H. (1989). Multilayer feedforward networks are universal approximators. *Neural Networks*, 2:359–366.

Isaacs, P. M. and Cohen, M. F. (1987). Controlling dynamic simulation with kinematic constraints, behavior functions and inverse dynamics. In Stone, M. C., editor, *Computer Graphics (SIGGRAPH '87 Proceedings)*, volume 21, pages 215–224.

Jacobs, R., Jordan, M. I., Nowlan, S. J., and Hinton, G. E. (1991). Adaptive mixtures of local experts. *Neural Computation*, 3:79–87.

Jordan, M. I. (1988). Supervised learning and systems with excess degrees of freedom. Technical Report 88-27, University of Massachusetts, Computer and Information Sciences, Amherst, MA.

Jordan, M. I. and Jacobs, R. A. (1994). Hierarchical mixture of experts and the em algorithm. *Neural Computation*, 6(2):181–214.

Jordan, M. I. and Rumelhart, D. E. (1992). Supervised learning with a distal teacher. *Cognitive Science*, 16:307–354.

Kass, M. and Miller, G. (1990). Rapid, stable fluid dynamics for computer graphics. In Baskett, F., editor, *Computer Graphics (SIGGRAPH '90 Proceedings)*, volume 24, pages 49–57.

Kawato, M., Setoyama, T., and Suzuki, R. (1988). Feedback error learning of movement by multi-layer neural network. In *Proceedings of the International Neural Networks Society First Annual Meeting.*

Kuperstein, M. (1987). Adaptive visual-motor coordination in multi-joint robots using parallel architecture. In *IEEE International Conference on Robotics and Automation*, pages 1595–1602.

Lamouret, A. and van de Panne, M. (1996). Motion synthesis by example. In *Proceedings of the 7th Eurographics Workshop on Simulation and Animation*, pages 199–212, Poitiers, France. Springer Verlag.

Lapedes, A. and Farber, R. (1987). Nonlinear signal processing using neural networks. Technical Report LA-UR-87-2662, Los Alamos National Laboratory, Los Alamos, NM.

Lasseter, J. (1987). Principles of traditional animation applied to 3D computer animation. In Stone, M. C., editor, *Computer Graphics (SIGGRAPH '87 Proceedings)*, volume 21, pages 35–44.

LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., and Jackel, L. (1989). Backpropagation applied to handwritten zip code recognition. *Neural Computation*, 1(4):541–551.

Lee, Y., Terzopoulos, D., and Waters, K. (1995). Realistic face modeling for animation. In Cook, R., editor, *SIGGRAPH 95 Conference Proceedings*, Annual Conference Series, pages 55–62. ACM SIGGRAPH, Addison Wesley. held in Los Angeles, California, 06-11 August 1995.

Liu, Z., Gortler, S. J., and Cohen, M. F. (1994). Hierarchical spacetime control. In Glassner, A., editor, *Proceedings of SIGGRAPH '94 (Orlando, Florida, July 24–29, 1994)*, Computer Graphics Proceedings, Annual Conference Series, pages 35–42. ACM SIGGRAPH, ACM Press. ISBN 0-89791-667-0.

Luenberger, D. G. (1984). *Linear and Nonlinear Programming.* Addison-Wesley, Reading, MA, second edition.

McKenna, M. and Zeltzer, D. (1990). Dynamic simulation of autonomous legged locomotion. In Baskett, F., editor, *Computer Graphics (SIGGRAPH '90 Proceedings)*, volume 24, pages 29–38.

Mendel, J. M. and McLaren, R. W. (1970). Reinforcement learning control and pattern recognition systems. In Mendel, J. M. and Fu, K. S., editors, *Adaptive, learning and pattern recognition systems: Theory and applications*, pages 287–318. Academic Press, New York.

Miller, G. S. P. (1988). The motion dynamics of snakes and worms. In Dill, J., editor, *Computer Graphics (SIGGRAPH '88 Proceedings)*, volume 22, pages 169–178.

Miller, W. T., Glanz, F. H., and Kraft, L. G. (1987). Applications of a general learning algorithm to the control of robotic manipulators. *International Journal of Robotics Research*, 6:84–96.

Minsky, M. L. and Papert, S. A. (1969). *Perceptrons.* MIT Press, Cambridge, MA. Expanded Edition 1990.

Moody, J. and Darken, C. J. (1989). Fast learning in networks of locally-tuned processing units. *Neural Computation*, 1:281–294.

Narendra, K. S. and Parthasarathy, K. (1991). Gradient methods for the optimization of dynamical systems containing neural networks. *IEEE Transactions on Neural Networks*, 2(2):252–262.

Ngo, J. T. and Marks, J. (1993). Spacetime constraints revisited. In Kajiya, J. T., editor, *Computer Graphics (SIGGRAPH '93 Proceedings)*, volume 27, pages 343–350.

Nguyen, D. and Widrow, B. (1989). The truck backer-upper: An example of self-learning in neural networks. In *Proceedings of the International Joint Conference on Neural Networks*, volume 2, pages 357–363. IEEE Press.

Pandy, M. G., Anderson, F. C., and Hull, D. G. (1992). A parameter optimization approach for the optimal control of large-scale musculoskeletal systems. *Transactions of the ASME*, 114(450).

Parker, D. B. (1985). Learning logic. Technical Report TR-47, MIT Center for Research in Computational Economics and Menagement Science, Cambridge, MA.

Perrone, M. P. (1993). General averaging results for convex optimization. In Mozer, M. C., editor, *Proceedings 1993 Connectionist Models Summer School*, pages 364–371, Hillsdale, NJ. Lawrence Erlbaum.

Perrone, M. P. and Cooper, L. N. (1994). When networks disagree: ensamble methods for hybrid neural networks. In Mammone, R. J., editor, *Artificial Neural Networks for Speech and Vision*, pages 126–142. Chapman and Hall, London.

Platt, J. C. and Barr, A. H. (1988). Constraint methods for flexible models. In Dill, J., editor, *Computer Graphics (SIGGRAPH '88 Proceedings)*, volume 22, pages 279–288.

Polak, E. (1971). *Computational Methods for Optimization: A Unified Approach.* Academic Press, New York.

Raibert, M. H. and Hodgins, J. K. (1991). Animation of dynamic legged locomotion. In Sederberg, T. W., editor, *Computer Graphics (SIGGRAPH '91 Proceedings)*, volume 25, pages 349–358.

Ridsdale, G. (1990). Connectionist modeling of skill dynamics. *Journal of Visualization and Computer Animation*, 1(2):66–72.

Rosenblatt, F. (1962). *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms.* Spartan, Washington, DC.

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). Learning internal representations by error backpropagation. In Rumelhart, D. E., McCleland, J. L., and the PDP Research Group, editors, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, volume 1, pages 318–362. MIT Press.

Shoemake, K. (1985). Animating rotation with quaternion curves. In Barsky, B. A., editor, *Computer Graphics (SIGGRAPH '85 Proceedings)*, volume 19, pages 245–254.

Sims, K. (1994). Evolving virtual creatures. In Glassner, A., editor, *Proceedings of SIGGRAPH '94 (Orlando, Florida, July 24–29, 1994)*, Computer Graphics Proceedings, Annual Conference Series, pages 15–22. ACM SIGGRAPH, ACM Press. ISBN 0-89791-667-0.

Stam, J. and Fiume, E. (1993). Turbulent wind fields for gaseous phenomena. In Kajiya, J. T., editor, *Computer Graphics (SIGGRAPH '93 Proceedings)*, volume 27, pages 369–376.

Stewart, A. J. and Cremer, J. F. (1992). Beyond keyframing: An algorithmic approach to animation. In *Proceedings of Graphics Interface '92*, pages 273–281.

Sutton, R. S. (1984). *Temporal credit assignment in reinforcement learning.* PhD thesis, University of Massachusetts, Amherst.

Terzopoulos, D. and Fleischer, K. (1988). Deformable models. *The Visual Computer*, 4(6):306–331.

Terzopoulos, D., Platt, J., Barr, A., and Fleischer, K. (1987). Elastically deformable models. In Stone, M. C., editor, *Computer Graphics (SIGGRAPH '87 Proceedings)*, volume 21, pages 205–214.

Terzopoulos, D. and Waters, K. (1990). Physically-based facial modelling, analysis, and animation. *Visualization and Computer Animation*, 1:73–80.

Tu, X. and Terzopoulos, D. (1994). Artificial fishes: Physics, locomotion, perception, behavior. In Glassner, A., editor, *Proceedings of SIGGRAPH '94 (Orlando, Florida, July 24–29, 1994)*, Computer Graphics Proceedings, Annual Conference Series, pages 43–50. ACM SIGGRAPH, ACM Press. ISBN 0-89791-667-0.

Unuma, M., Anjyo, K., and Takeuchi, R. (1995). Fourier principles for emotion-based human figure animation. In Cook, R., editor, *SIGGRAPH 95 Conference Proceedings*, Annual Conference Series, pages 91–96. ACM SIGGRAPH, Addison Wesley. held in Los Angeles, California, 06-11 August 1995.

van de Panne, M. (1996). Parameteterized gait synthesis. In *IEEE Computer Graphics And Applications*, volume 16, pages 40–49.

van de Panne, M. and Fiume, E. (1993). Sensor-actuator networks. In Kajiya, J. T., editor, *Computer Graphics (SIGGRAPH '93 Proceedings)*, volume 27, pages 335–342.

Weigend, A. S. and Gershenfeld, N. A. (1994). Time series prediction: Forecasting the future and understanding the past. In *Santa Fe Institute Studies in the Science of Complexity.* Santa Fe Institute.

Wejchert, J. and Haumann, D. (1991). Animation aerodynamics. In Sederberg, T. W., editor, *Computer Graphics (SIGGRAPH '91 Proceedings)*, volume 25, pages 19–22.

Werbos, P. J. (1974). *Beyond regression: new tools for prediction and analysis in the behavioral sciences.* PhD thesis, Harvard University, Boston, MA.

Werbos, P. J. (1988). Generalization of back propagation with applications to a recurrent gas market model. *Neural Networks*, 1:339–356.

Werbos, P. J. (1990). A menu of designs for reinforcement learning over time. In Miller, W. T., Sutton, R. S., and Werbos, P. J., editors, *Neural Networks for Control*, pages 67–95. The MIT Press.

Widrow, B. (1986). Adaptive inverse control. In *Second IFAC Workshop on Adaptive Systems in Control and Signal Processing*, pages 1–5, Lund, Sweden. Lund Institute of Technology.

Widrow, B. and Lehr, M. A. (1990). 30 years of adaptive neural networks: perceptron, madeline, and backpropagation. In *Proceedings of the IEEE*, volume 78, pages 1415–1442.

Widrow, B., McCool, J., and Medoff, B. (1978). Adaptive control by inverse modeling. In *Twelfth Asilomar Conference on Circuits, Systems, and Computers.*

Wilhelms, J. (1987). Using dynamic analysis for realistic animation of articulated bodies. *IEEE Computer Graphics and Applications*, 7(6):12–27.

Williams, R. J. (1988). On the use of backpropagation in associative reinforcement learning. In *Proceedings of the IEEE International Conference on Neural Networks*, pages 263–270, San Diego, CA.

Witkin, A. and Kass, M. (1988). Spacetime constraints. In Dill, J., editor, *Computer Graphics (SIGGRAPH '88 Proceedings)*, volume 22, pages 159–168.

Witkin, A. and Popović, Z. (1995). Motion warping. In Cook, R., editor, *SIGGRAPH 95 Conference Proceedings*, Annual Conference Series, pages 105–108. ACM SIGGRAPH, Addison Wesley. held in Los Angeles, California, 06-11 August 1995.

Witkin, A. and Welch, W. (1990). Fast animation and control of nonrigid structures. In Baskett, F., editor, *Computer Graphics (SIGGRAPH '90 Proceedings)*, volume 24, pages 243–252.

Yu, Q. (1998). Synthetic motion capture for interactive virtual worlds. Master's thesis, University of Toronto.