From: AAAI Technical Report SS-99-05. Compilation copyright © 1999, AAAI (www.aaai.org). All rights reserved.

Fast Neural Network Emulation and Control of Dynamical Systems

Radek Grzeszczuk [1] **Demetri Terzopoulos** [2] **Geoffrey Hinton** [2]

[1] Intel Corporation
Microcomputer Research Lab
2200 Mission College Blvd.
Santa Clara, CA 95052, USA

[2] University of Toronto
Department of Computer Science
10 King's College Road
Toronto, ON M5S 3H5, Canada

Abstract

Computer animation through the numerical simulation of physics-based graphics models offers unsurpassed realism, but it can be computationally demanding. This paper demonstrates the possibility of replacing the numerical simulation of nontrivial dynamic models with a dramatically more efficient "NeuroAnimator" 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. We demonstrate NeuroAnimators for a variety of physics-based models. By exploiting the network structure of the NeuroAnimator, we also introduce a remarkably fast algorithm for learning controllers that enables either complex physics-based models or their neural network emulators to synthesize motions satisfying prescribed animation goals.

Introduction

Animation based on physical principles has been an influential trend in computer graphics for over a decade (see, e.g., (1; 2; 3)). This is not only due to the unsurpassed realism that physics-based techniques offer. In conjunction with suitable control and constraint mechanisms, physical models also facilitate the production of copious quantities of realistic animation in a highly automated fashion. Physics-based animation techniques are beginning to find their way into high-end commercial systems. However, a well-known drawback has retarded their broader penetration—compared to geometric models, physical models typically entail formidable numerical simulation costs. Furthermore, a hurdle, known as the "physics-based animation control problem", is that of computing the control forces such that the physical model produces motions that satisfy the goals specified by the animator.

This paper 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*. Exploiting the network structure of the NeuroAnimator, we also introduce a remarkably fast algorithm for learning controllers that enables either complex physics-based models or their neural network emulators to synthesize motions satisfying prescribed animation goals.

A popular approach to the animation control problem is *controller synthesis* (4; 5; 6). Controller synthesis is a generate-and-test strategy. Through repeated forward simulation of the physics-based model, controller synthesis 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.

We demonstrate that the NeuroAnimator enables a novel, highly efficient approach to controller synthesis. Outstanding efficiency results not only because of fast controller evaluation through NeuroAnimator emulation of the dynamics of the physical model. To a large degree it also stems from the fact that we can exploit the 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. NeuroAnimator controllers are equally applicable to controlling the original physics-based models.

Our work is inspired in part by that of Nguyen and Widrow (7). Their "truck backer-upper" demonstrated the neural network based approximation and control of a nonlinear kinematic system. We introduce several generalizations that enable us to tackle a variety of complex, fully dynamic models in the context of computer animation. Connectionist approximations of dynamical systems have been also applied to robot control (see, e.g., (8; 9)).

The NeuroAnimator Approach

Our approach is motivated by the following considerations: Whether we are dealing with rigid (2), articulated (3), or nonrigid (1) 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 $\mathbf{s}_{t+\delta t} = \Phi[\mathbf{s}_t, \mathbf{u}_t, \mathbf{f}_t]$. 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 time $t + \delta t$ 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 requires the evaluation of the map Φ at every timestep, which usually involves a nontrivial 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 $O(N^3)$ per timestep.

Is it possible to replace the conventional numerical simulator 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 offer a general mechanism for approximating complex maps in higher dimensional spaces (10).[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 graphics 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. 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 the discrete-time dynamical system. 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.

From Physics-Based Models to NeuroAnimators

Our task is to construct neural networks that approximate Φ in the dynamical system. We propose to employ backpropagation to train feedforward networks $\mathbf{N}_\Phi$, with a single layer of sigmoidal hidden units, to predict future states using super timesteps $\Delta t = n\delta t$ while containing the approximation error so as not to appreciably degrade the physical realism of the resulting animation. The basic emulation step is $\mathbf{s}_{t+\Delta t} = \mathbf{N}_\Phi[\mathbf{s}_t, \mathbf{u}_t, \mathbf{f}_t]$. 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. The emulation process is a sequence of these evaluations. 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. Since the emulation step is large compared with the numerical simulation step, we resample the motion trajectory at the animation frame rate, computing intermediate states through linear interpolation of states obtained from the emulation.

Network Input/Output Structure

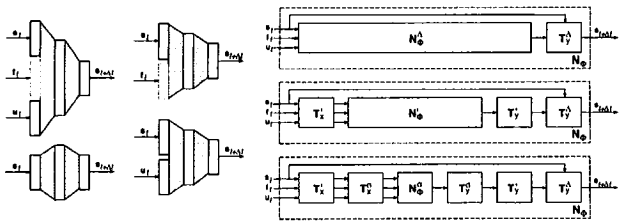
Fig. 1(a) illustrates different emulator input/output structures. The emulator network has a single set of output variables specifying $\mathbf{s}_{t+\Delta t}$. In general, for a so-called active model, which includes control inputs, under the influence of unpredictable applied forces, we employ a full 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, thus yielding a simpler emulator with two input sets $\mathbf{s}_t$ and $\mathbf{u}_t$. The simplest type of emulator has only a single set of inputs $\mathbf{s}_t$. This emulator suffices to approximate passive models acted upon by deterministic external forces.

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.

A fundamental problem is that the state variables of a dynamical system can have a large dynamic range (in principle, from $-\infty$ to $+\infty$). 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. 1(b) (top) 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.

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



(a) (b)

Figure 1: (a) Different types of emulators. (b) 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 and (11) for the details).

We can further improve the approximation power of the emulator network by exploiting natural invariances. In particular, since the map Φ is invariant under rotation and translation, 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 (Fig. 1(b) (center)).

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. 1(b)) (bottom) we replace $\mathbf{N}_\Phi'$ with an operator $\mathbf{T}_x^\sigma$ that normalizes its inputs, a network $\mathbf{N}_\Phi^\sigma$ that assumes 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. 1(b) is structurally more complex than the standard feedforward neural network $\mathbf{N}_\Phi$ that it replaces, the operators denoted by $\mathbf{T}$ 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.

Hierarchical Networks

As a universal function approximator, a neural network should in principle be able to approximate the map Φ for any dynamical system, given enough sigmoid hidden units and training data. In practice, however, the number of hidden layer neurons needed and the training data requirements grow quickly with the size of the network, often making the training of large networks impractical. To overcome the "curse of dimensionality," we have found it prudent to structure NeuroAnimators for all but the simplest physics-based models as hierarchies of smaller networks rather than as large, monolithic networks. The strategy behind a hierarchical 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 network.

Training NeuroAnimators

To arrive at a NeuroAnimator for a given physics-based model, we train the constituent neural network(s) through backpropagation 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, once a NeuroAnimator is trained offline, it can be reused online to produce an infinite variety of fast animations. The important point is that by generalizing from the sparse training examples, 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. To generate each training example, we could 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 good neural network approximation $\mathbf{N}_\Phi$ of the map Φ, we would like ideally to sample Φ as uniformly as possible over its domain, with randomly chosen initial conditions among all valid state, external force, and control combinations. However, we can make better use of computational resources by sampling those state, force, and control inputs that typically occur as a physics-based model is used in practice.

We employ a neural network simulator called *Xerion* which was developed at the University of Toronto. 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 significantly. We use the conjugate gradient method to train networks of small and moderate size. For large networks, we use gradient descent with momentum. We divide the training examples into mini-batches, each consisting of approximately 30 uncorrelated examples, and update the network weights after processing each mini-batch.

Emulation Results

We have successfully constructed and trained several NeuroAnimators to emulate a variety of physics-based models:

(1) a physics-based model of a planar multi-link pendulum suspended in gravity, subject to joint friction forces, external forces applied on the links, and controlled by independent motor torques at each of the three joints, (2) a physics-based model of a truck implemented as a rigid body, subject to friction forces where the tires contact the ground, controlled by rear-wheel drive (forward and reverse) and steerable front wheels, (3) a physics-based model of a lunar lander, implemented as a rigid body subject to gravitational forces and controlled by a main rocket thruster and three independent attitude jets, and (4) a biomechanical (mass-spring-damper) model of a dolphin capable of swimming in simulated water via the coordinated contraction of 6 independently controlled muscle actuators which deform its body, producing hydrodynamic propulsion forces.

We used SD/FAST (a rigid body dynamics simulator marketed by Symbolic Dynamics, Inc.) to simulate the dynamics of the rigid body and articulated models, and we employ the simulator developed in (13) to simulate the deformable-body dynamics of the dolphin.

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 sigmoidal hidden units, but rather used enough units 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.

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. By comparison, a single simulation timestep using an implicit time integration scheme requires $O(N^3)$ operations. Moreover, a forward pass through the neural network is often equivalent to as many as 50 physical simulation steps, so the efficiency is even more dramatic, yielding performance improvements up to two orders of magnitude faster than the physical simulator. A NeuroAnimator that predicts 100 physical simulation steps offers a speedup of anywhere between 50 and 100 times depending on the type of physical model.

Control Learning

An additional benefit of the NeuroAnimator is that it enables a novel, highly efficient approach to the difficult problem of controlling physics-based models to synthesize motions that satisfy prescribed animation goals. The neural network approximation to the physical model is differentiable; hence, it can be used to discover the causal effects that control force inputs have on the actions of the models. Outstanding efficiency stems from exploiting the trained NeuroAnimator to compute partial derivatives of output states with respect to control inputs. The efficient computation of the approximate gradient enables the utilization of fast gradient-based optimization for controller synthesis. Nguyen and Widrow's (7) "truck backer-upper" demonstrated the neural network based approximation and control of a nonlinear kinematic system. Our technique offers a new controller synthesis algorithm that works well in dynamic environments with changing control objectives. See (11; 12) for the details.

In the following sections, 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 backpropagation algorithm.

Objective Function and Optimization

We write a sequence of emulation steps as

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

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.

Following the control learning formulation in (6), we define a discrete objective function

$$J(\mathbf{u}) = \mu_u J_u(\mathbf{u}) + \mu_s J_s(\mathbf{s}), \tag{2}$$

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 NeuroAnimator using $\mathbf{u}$, according to (1). Via the controller evaluation term J_u, we may wish to promote a preference for controllers with certain desirable qualities, such as smooth lower amplitude controllers. 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

$$J(\mathbf{u}) = \frac{\mu_u}{2} \sum_{i=1}^{M} \mathbf{u}_i^2 + \frac{\mu_s}{2} (\mathbf{s}_{M+1} - \mathbf{s}_d)^2, \tag{3}$$

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.

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 (2). As mentioned earlier, we exploit the NeuroAnimator structure to arrive at an efficient gradient descent optimizer:

$$\mathbf{u}^{l+1} = \mathbf{u}^l + \eta_x \nabla_{\mathbf{u}} J(\mathbf{u}^l), \tag{4}$$

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

At each iteration l, the algorithm first emulates the forward dynamics according to (1) using the control inputs

$\mathbf{u}^l = [\mathbf{u}_1^l, \mathbf{u}_2^l, \ldots, \mathbf{u}_M^l]$ to yield the motion sequence $\mathbf{s}^l = [\mathbf{s}_1^l, \mathbf{s}_2^l, \ldots, \mathbf{s}_{M+1}^l]$. Next, it computes the components of $\nabla_{\mathbf{u}} J$ in (4) in an efficient manner using *backpropagation through time* (14). Instead of adjusting weights as in normal backpropagation, however, the algorithm adjusts neuronal inputs, specifically, the control inputs. It thus proceeds in reverse through the network cascade computing components of the gradient. Fig. 1(b) illustrates the backpropagation through time process, showing the sequentially computed controller updates $\delta\mathbf{u}_M$ to $\delta\mathbf{u}_0$.

The forward emulation and control adjustment steps are repeated for each iteration of (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 descent towards an optimum is possible because the trained NeuroAnimator provides a gradient direction.

The control algorithm based on the differentiation of the emulator of the forward model has 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 re-learns very quickly.

An additional advantage of our 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.

Controller Learning Results

We have successfully applied our backpropagation through time controller learning algorithm to the presented NeuroAnimators. We find the technique very effective—it routinely computes solutions to non-trivial control problems in just a few iterations. The efficiency of the fast convergence rate is further amplified by the replacement of costly physical simulation with much faster NeuroAnimator emulation. These two factors yield outstanding speedups, as we report below.

Fig. 2(a) shows the progress of the control learning algorithm for the 3-link pendulum. The purple pendulum, animated by a NeuroAnimator, is given the goal to end the animation with zero velocity in the position indicated in green. 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 backpropagation through time iterations to learn a successful controller.

Fig. 2(b) 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. 2(c) 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 descent velocity prior to landing should be small in order to land softly. A successful landing controller was computed in 15 learning iterations.

Fig. 2(d) shows the dolphin NeuroAnimator learning to swim forward. The simple objective of moving as far forward as possible produces a natural, sinusoidal swimming pattern.

All trained controllers have a duration of 20 seconds of animation time; i.e., they take the equivalent of 2,000 physical simulation timesteps, or 40 emulator super-timesteps using $\mathbf{N}_{\Phi}^{50}$ emulator. The number of control variables (M in $\mathbf{u} = [\mathbf{u}_1, \mathbf{u}_2, \ldots, \mathbf{u}_M]$) 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 truck 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 6 muscle actuators.

To give an idea of actual running times, the synthesis of the swimming controller which took more than 1 hour using the technique in (6) now takes less than 10 seconds on the same computer.

Conclusion

We have introduced an efficient alternative to the conventional approach of producing physically realistic animation through numerical simulation. Our approach involves the learning of neural network emulators of physics-based models by observing the dynamic state transitions produced by such models in action. The emulators approximate physical dynamics with dramatic efficiency, yet without serious loss of apparent fidelity. Our performance benchmarks indicate that the neural network emulators can yield physically realistic animation one or two orders of magnitude faster than conventional numerical simulation of the associated physics-based models. Our new control learning algorithm, which exploits fast emulation and the differentiability of the network approximation, is orders of magnitude faster than competing controller synthesis algorithms for computer animation.

References

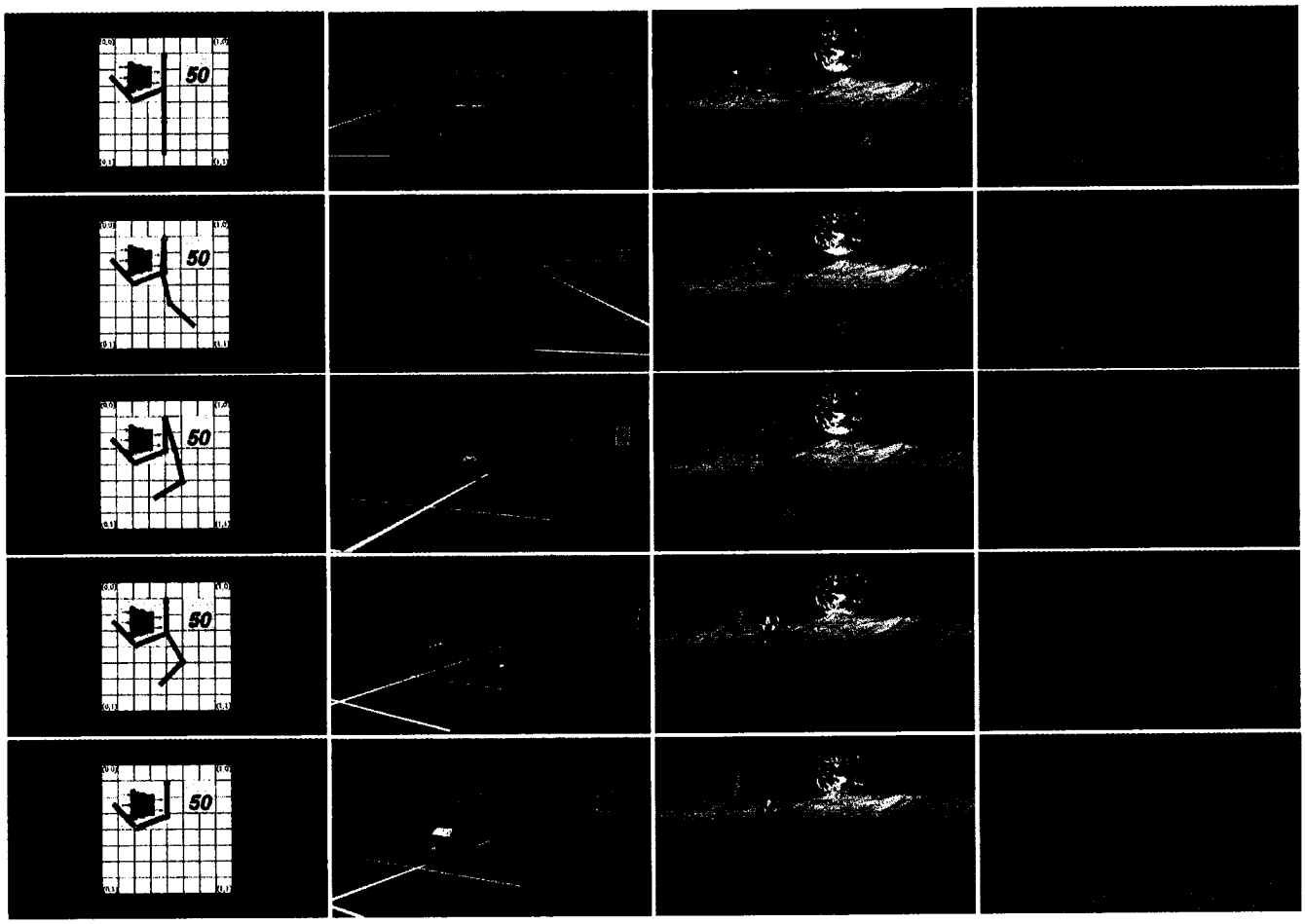
[1] D. Terzopoulos, J. Platt, A. Barr, K. Fleischer. Elastically deformable models. In M.C. Stone, ed., *Computer Graphics (SIGGRAPH '87 Proceedings)*, **21**, 205–214, July 1987.

[2] J.K. Hahn. Realistic animation of rigid bodies. In J. Dill, ed., *Computer Graphics (SIGGRAPH '88 Proceedings)*, **22**, 299–308, August 1988.

[3] J.K. Hodgins, W.L. Wooten, D.C. Brogan, J.F. O'Brien. Animating human athletics. In R. Cook, ed., *Proc. of ACM SIGGRAPH 95 Conf.*, 71–78, August, 1995.

[4] J.T. Ngo and J. Marks. Spacetime constraints revisited. *Proc. of ACM SIGGRAPH 93 Conf.*, 343–350, August 1993.

[5] M. van de Panne and E. Fiume. Sensor-actuator networks. *Proc. of ACM SIGGRAPH 93*, 335–342, August 1993.



(a) (b) (c) (d)

Figure 2: Results of applying the control learning algorithm to four different NeuroAnimators. (a) The pendulum NeuroAnimator shown in purple needs to reach the state indicated by the green pendulum, with zero final velocity. (b) The truck NeuroAnimator learning to park in the position and orientation indicated by the translucent vehicle in the background. (c) The lunar lander NeuroAnimator learning to land in the position and orientation of the translucent vehicle sitting on the ground, with minimal descent velocity. (d) The dolphin NeuroAnimator learning to swim. The objective of locomoting as far forward as possible produces a natural, periodic swimming pattern.

[6] R. Grzeszczuk and D. Terzopoulos. Automated learning of muscle-actuated locomotion through control abstraction. *Proc. SIGGRAPH 95 Conf.*, 63–70, August 1995.

[7] D. Nguyen, B. Widrow. The truck backer-upper: An example of self-learning in neural networks. In *Proc. Inter. Joint Conf. Neural Networks*, 357–363. IEEE Press, 1989.

[8] M.I. Jordan. Supervised learning and systems with excess degrees of freedom. Technical Report 88-27, Univ. of Massachusetts, Comp.& Info. Sci., Amherst, MA, 1988.

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

[10] G. Cybenko. Approximation by superposition of sigmoidal function. *Math. of Control Signals & Systems*, **2**(4):303–314, 1989.

[11] R. Grzeszczuk. *NeuroAnimator: Fast Neural Network Emulation and Control of Physics-Based Models*. PhD thesis, Dept. of Comp. Sci., Univ. of Toronto, May 1998.

[12] R. Grzeszczuk, D. Terzopoulos, G. Hinton. NeuroAnimator: Fast neural network emulation and control of physics-based models. *Proc. of ACM SIGGRAPH 98 Conf.*, 9–20, July 1998.

[13] X. Tu, D. Terzopoulos. Artificial fishes: Physics, locomotion, perception, behavior. In A. Glassner, ed., *Proc. of ACM SIGGRAPH 94 Conf.*, 43–50. July 1994.

[14] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error backpropagation. In D. E. Rumelhart et al. (eds.), *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, vol. 1, pages 318–362. MIT Press, 1986.