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

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

[1] Intel Corporation [2] University of Toronto

Abstract: Animation through the numerical simulation of physics-based graphics models offers unsurpassed realism, but it can be computationally demanding. Likewise, the search for 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 dynamic models 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 a variety of physics-based models.

CR Categories: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Animation; I.6.8 [Simulation and Modeling]: Types of Simulation—Animation

Keywords: physics-based animation, neural networks, learning, motion control, backpropagation, dynamical systems, simulation.

1 Introduction

Animation based on physical principles has been an influential trend in computer graphics. 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.

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*. The network structure of NeuroAnimators furthermore enables a new solution to the control problem associated with physics-based models, leading to a remarkably fast algorithm for synthesizing motions that satisfy prescribed animation goals.

1.1 Overview of the NeuroAnimator Approach

Our approach is motivated by the following considerations: Whether we are dealing with rigid [6, 1], articulated [7, 19], or nonrigid [17, 10] 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

$$\mathbf{s}_{t+\delta t} = \Phi[\mathbf{s}_t, \mathbf{u}_t, \mathbf{f}_t]. \tag{1}$$

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 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 (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 pose 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 [2] offer a general mechanism for approximating complex maps in higher dimensional spaces.[2] Our premise is that, to a sufficient degree of accuracy and at significant computational savings, trained neural networks can approximate maps Φ

[1] 2200 Mission College Blvd., Santa Clara, CA 95052, RN6-35
E-mail: `radek.grzeszczuk@intel.com`
[2] 10 King's College Road, Toronto, Ontario, Canada, M5S 3G4
E-mail: {`dt`|`hinton`}`@cs.toronto.edu`

Published in the Proceedings of SIGGRAPH 98 (Orlando, FL, July 19–24, 1998). In *Computer Graphics* Proceedings, Annual Conference Series, 1998, ACM SIGGRAPH, pp. 9–20.

[2] Note that Φ in (1) 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.

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. 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). 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), its neural network approximation is analytically differentiable. In fact, the derivative of NeuroAnimator state outputs with respect to control and external force inputs is efficiently computable by applying the chain rule of differentiation. Easy differentiability enables us to arrive at a remarkably fast gradient descent optimization algorithm to compute optimal or near-optimal controllers. These controllers produce a series of control inputs $\mathbf{u}_t$ that enable NeuroAnimators to synthesize motions satisfying prescribed constraints on the desired animation. NeuroAnimator controllers are equally applicable to controlling the original physics-based models.

1.2 Related Work

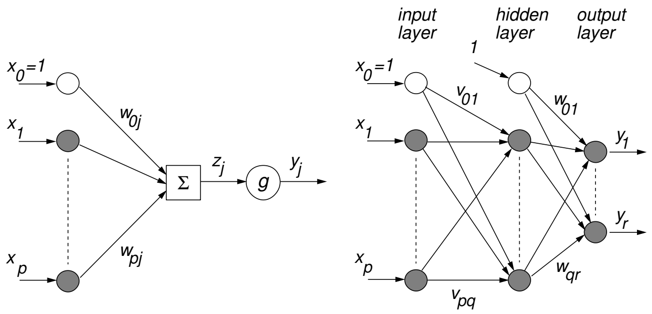
To date, network architectures have found only a few applications in computer graphics. One application has been the control of animated characters. Ridsdale [14] reports a method for skill acquisition using a connectionist model of skill memory. The sensor-actuator networks of van de Panne and Fiume [19] are recurrent networks of units that take sensory information as input and produce actuator controls as output. Sims [16] employed a network architecture to structure simple "brains" that control evolved creatures. Our work differs fundamentally from these efforts.

The basis of our approach is related to work presented in the mainstream neural network literature on connectionist control of complex systems. Nguyen and Widrow demonstrated the neural network based approximation and control of a nonlinear kinematic system in their "truck backer-upper" [12]. More recently, Jordan and Rumelhart [9] proposed a two step approach to learning controllers for physical robots. In step one, a neural net learns a predictive internal model of the robot, which maps from actions to state transitions. In step two this forward model is used to learn an inverse model that maps from intentions to actions, by training the inverse model so that it produces an identity transformation in cascade with the established forward model.

Inspired by these results, we exploit neural networks to produce controlled, physically realistic animation satisfying user-specified constraints at a fraction of the computational cost of conventional numerical simulation.

2 Artificial Neural Networks

In this section we define a common type of artificial neural network and describe the backpropagation training algorithm. NeuroAnimator makes use of a neural network simulator called *Xerion* which was developed at the University of Toronto and is available publicly.[3] The public availability of software such as Xerion contributes to making our NeuroAnimator approach easily accessible to the graphics community.



(a) (b)

Figure 1: (a) Mathematical model of a neuron j. (b) Three-layer feedforward neural network $\mathbf{N}$. Bias units are not shaded.

2.1 Neurons and Neural Networks

In mathematical terms, a *neuron* is an operator that maps $\Re^p \mapsto \Re$. Referring to Fig. 1(a), neuron j receives a signal z_j that is the sum of p inputs x_i scaled by associated connection weights w_{ij}:

$$z_j = w_{0j} + \sum_{i=1}^{p} x_i w_{ij} = \sum_{i=0}^{p} x_i w_{ij} = \mathbf{x}^T \mathbf{w}_j, \qquad (2)$$

where $\mathbf{x} = [x_0, x_1, \ldots, x_p]^T$ is the input vector (the superscript T denotes transposition), $\mathbf{w}_j = [w_{0j}, w_{1j}, \ldots, w_{pj}]^T$ 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. 1(b) 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 $h_j = \sigma(\sum_{i=0}^{p} x_i v_{ij})$.

2.2 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

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

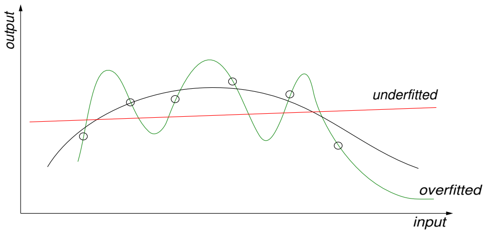


Figure 2: Depicted in a low-dimensional setting, a neural network with too few neurons underfits the training data. One with too many neurons overfits the data. The solid curve represents a properly chosen network which provides a good compromise between approximation (fits the training data) and generalization (generates reasonable output values away from the training examples).

a compact domain $\mathbf{x} \in \mathcal{X}$ [3, 8]; i.e., for an arbitrarily small $\epsilon > 0$ there exists a network $\mathbf{N}$ such that

$$\forall \mathbf{x} \in \mathcal{X}, \quad e(\mathbf{x}, \mathbf{w}) = \|\Phi(\mathbf{x}) - \mathbf{N}(\mathbf{x}, \mathbf{w})\|^2 < \epsilon, \tag{3}$$

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

$$\begin{cases} \mathbf{x}^\tau = [x_1^\tau, x_2^\tau, \ldots, x_p^\tau]^T; \\ \mathbf{y}^\tau = \Phi(\mathbf{x}^\tau) = [y_1^\tau, y_2^\tau, \ldots, y_r^\tau]^T, \end{cases} \tag{4}$$

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).

Training a neural network to approximate a map 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. Fig. 2 depicts these problems in a low-dimensional setting. To avoid underfitting, 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.

2.3 Backpropagation Learning Algorithm

Rumelhart, Hinton and Williams [15] proposed an efficient algorithm for training multi-layer feedforward networks, called the *backpropagation algorithm*. The backpropagation algorithm seeks to minimize the objective function

$$E(\mathbf{w}) = \sum_{\tau=1}^{n} e(\mathbf{x}^\tau, \mathbf{w}) = \sum_{\tau=1}^{n} E^\tau(\mathbf{w}) \tag{5}$$

which sums the approximation errors e from (3) over the n training examples. The off-line training version of the algorithm adjusts the weights of the network using the gradient descent formula

$$\mathbf{w}^{l+1} = \mathbf{w}^l + \eta \nabla_{\mathbf{w}} E(\mathbf{w}^l), \tag{6}$$

where $\nabla_{\mathbf{w}} E$ denotes the gradient of the objective function with respect to the weights and $\eta < 1$ is referred to as the *learning rate*.

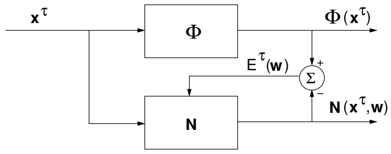


Figure 3: The backpropagation algorithm learns a map Φ by adjusting the weights $\mathbf{w}$ of the network $\mathbf{N}$ in order to reduce the difference between in the network output $\mathbf{N}(\mathbf{x}^\tau, \mathbf{w})$ and the desired output $\Phi(\mathbf{x}^\tau)$. Depicted here is the on-line version of the algorithm that adjusts the weights of the network after observing each training example.

An on-line training version of the backpropagation algorithm that adjusts the weights of the network after each training example is presented in [15]. Fig. 3 illustrates this process.

Backpropagation refers to the practical, recursive method to calculate the component error derivatives of the gradient term in (6). Applying the chain rule of differentiation, the backpropagation algorithm first computes the derivatives with respect to weights in the output layer and chains its way back to the input layer, computing the derivatives with respect to weights in each hidden layer as it proceeds.

To improve the learning rate, the gradient descent rule of the basic backpropagation algorithm (6), which takes a fixed step in the direction of the gradient, can be replaced by more sophisticated nonlinear optimization techniques. Line search offers one way to accelerate the training by searching for the optimal size step in the gradient direction. Additional performance improvement can be achieved by taking at each optimization step a direction orthogonal to previous directions. This is known as the conjugate gradient method [13]. A simple but effective method for increasing the learning rate augments the gradient descent update rule with a *momentum term*. The momentum method updates the weights as follows:

$$\delta \mathbf{w}^{l+1} = -\eta_w \nabla_{\mathbf{w}} E(\mathbf{w}^l) + \alpha_w \delta \mathbf{w}^l, \tag{7}$$

$$\mathbf{w}^{l+1} = \mathbf{w}^l + \delta \mathbf{w}^{l+1}, \tag{8}$$

where the momentum parameter α_w must be between 0 and 1. The neural network simulator Xerion includes the above optimization techniques and several others. Later in the paper we discuss the types of optimization methods that we used to train NeuroAnimators and to synthesize controllers.

3 From Physics-Based Models to NeuroAnimators

In this section we explain the practical application of neural network concepts to the construction and training of different classes of NeuroAnimators. Among other subjects, this includes network input/output structure, the use of hierarchical networks to tackle physics-based models with large state spaces, and strategies for generating good training datasets. We also discuss the practical issue of applying the Xerion neural network simulator to train NeuroAnimators. Finally, we show sample results demonstrating the accurate emulation of various dynamic models.

Our task is to construct neural networks that approximate Φ in the dynamical system (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 physical realism of the resulting animation. Analogous to (1), the basic emulation step is

$$\mathbf{s}_{t+\Delta t} = \mathbf{N}_\Phi[\mathbf{s}_t, \mathbf{u}_t, \mathbf{f}_t]. \tag{9}$$

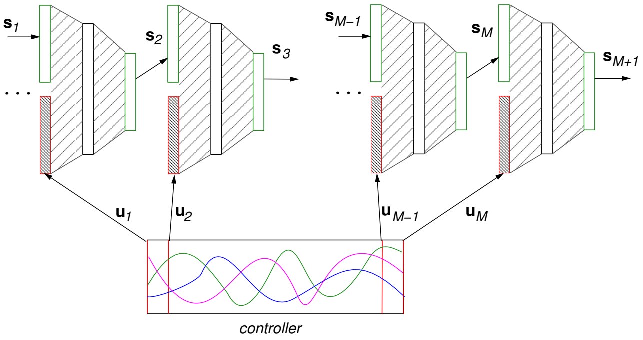


Figure 4: Forward emulation using a neural network. At each iteration, the network output becomes the state input at the next iteration of the algorithm. Note that the same emulator network is used recursively.

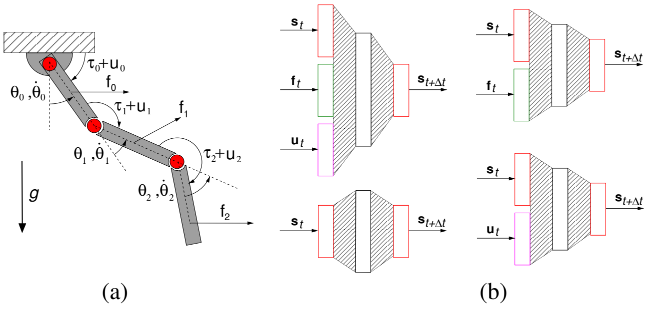


Figure 5: Three-link physical pendulum and network emulators. (a) An active pendulum with joint friction τ_i, motor torques u_i, applied forces $\mathbf{f}_i$, and gravity $\mathbf{g}$. Without motor torques, the pendulum is passive. (b) Different types of emulators.

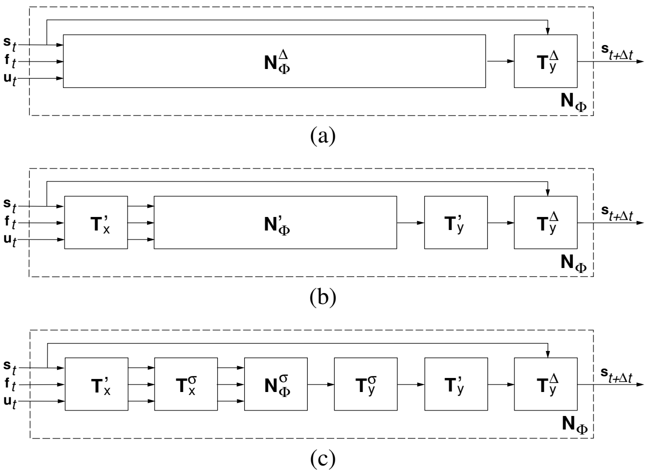


Figure 6: Transforming a simple feedforward neural network $\mathbf{N}_\Phi$ into a practical emulator network $\mathbf{N}^\sigma_\Phi$ that is easily trained to emulate physics-based models. The following operators perform the appropriate pre- and post-processing: $\mathbf{T}'_x$ transforms inputs to local coordinates, $\mathbf{T}^\sigma_x$ normalizes inputs, $\mathbf{T}^\sigma_y$ unnormalizes outputs, $\mathbf{T}'_y$ transforms outputs to global coordinates, $\mathbf{T}^\Delta_y$ converts from a state change to the next state (see text).

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. Fig. 4 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 employs a recurrent network whose outputs become inputs for 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 too coarse for animation. To avoid motion artifacts, we resample the motion trajectory at the animation frame rate, computing intermediate states through linear interpolation of states obtained from the emulation. Linear interpolation produces satisfactory motion, although a more sophisticated scheme could improve the result.

3.1 Network Input/Output 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), such as the multi-link pendulum illustrated in Fig. 5(a), with control inputs $\mathbf{u}$ comprising joint motor torques is known as active, otherwise, it is passive.

Fig. 5(b) illustrates different emulator input/output structures. 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. 5(a) is under the 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 need not be provided as 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 forces.

3.2 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. 6(a) we restructure our simple network $\mathbf{N}_\Phi$ as a network $\mathbf{N}^\Delta_\Phi$ 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. 6(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. Appendix A provides the mathematical details. 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. 6(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 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. 6(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. A more detailed presentation of the restructured emulator can be found in [4].

3.3 Hierarchical Networks

As a universal function approximator, a neural network should in principle be able to approximate the map Φ in (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 physics-based model.

In particular, neural networks are susceptible to the "curse of dimensionality". The number of neurons needed in hidden layers and the training data requirements grow quickly with the size of the network, often making the training of large networks impractical. 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.

A natural example of hierarchical networks arises when approximating complex articulated models, such as Hodgins' mechanical human runner model [7] which has a tree structure with a torso and limbs. Rather than collect all of its 30 controlled degrees of freedom into a single large network, it is natural to emulate the model using 5 smaller networks: a torso network plus left and right arm and leg networks.

Hierarchical representations are also useful when dealing with deformable models with large state spaces, such as the biomechanical model of a dolphin described in [5] which we use in our experiments. The mass-spring-damper dolphin model (Fig. 7) consists of

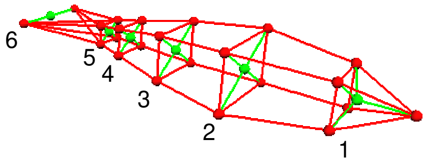


Figure 7: Hierarchical state representation for the dolphin mechanical model. Red nodes mark point masses and green nodes mark numbered local centers of mass. Green lines associate groups of point masses to their local center.

23 point masses, yielding a state space with $23 \times 3 = 69$ positions and 69 velocities, plus 6 controlled degrees of freedom—one for each independent actuator. 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.

3.4 Training NeuroAnimators

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. 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.

The initial conditions can be sampled at random among all valid state, external force, and control combinations. To learn a good neural network approximation $\mathbf{N}_{\Phi}$ of the map Φ in (1), we would like ideally to sample Φ as uniformly as possible over its domain. Unfortunately, for most physics-based models of interest, the domain has high dimensionality, often making a uniform sampling impractical. However, we can make the best use of computational resources by concentrating them on sampling those state, force, and control inputs that typically occur as a physics-based model is used in practice.

Fig. 8 illustrates an effective sampling strategy using the dynamic dolphin model as an example. 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

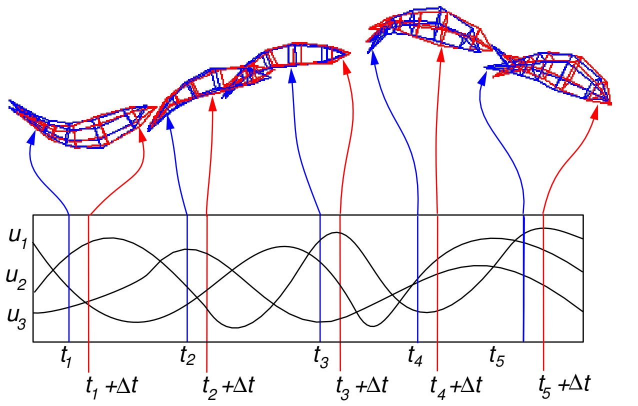


Figure 8: An effective state transition sampling strategy illustrated using the dynamic dolphin model. The dynamic model is simulated numerically with typical control input functions $\mathbf{u}$. For each training example generated, the blue model represents the input state (and/or control and external 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.

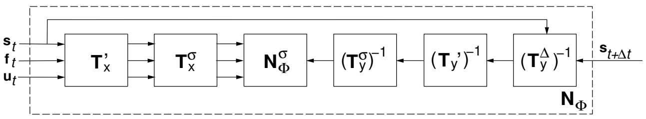


Figure 9: Transforming the training data for consumption by the network $\mathbf{N}^\sigma_\Phi$ in Fig. 6(c). The inputs of the training set are transformed through the operators on the input side of the network in Fig. 6(c). The outputs of the training set are transformed through the inverses of the operators at the output side of the network in Fig. 6(c).

record a set of training examples $\{[\mathbf{s}^T_{t_k}, \mathbf{f}^T_{t_k}, \mathbf{u}^T_{t_k}]^T; \mathbf{s}_{t_k+\Delta t}\}$, $k = 1, 2, \ldots$ 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.

3.5 Network Training in Xerion

As mentioned earlier, to train the emulator shown in Fig. 6(c) we need only train the network $\mathbf{N}^\sigma_\Phi$ because the operators denoted by the letter $\mathbf{T}$ are predetermined. As shown in Fig. 9, before presenting the training data to the network $\mathbf{N}^\sigma_\Phi$, we transform the inputs of the training set through the operators $\mathbf{T}'_x$ and $\mathbf{T}^\sigma_x$ and transform the associated outputs through the operators $(\mathbf{T}^\Delta_y)^{-1}$, $(\mathbf{T}'_y)^{-1}$, and $(\mathbf{T}^\sigma_\Phi)^{-1}$ which are the inverses of the corresponding operators used during the forward emulation step shown in Fig. 6(c).

We begin the off-line training process by initializing the weights of $\mathbf{N}^\sigma_\Phi$ 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) 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 (7–8) 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 B contains an example Xerion script which specifies and trains a NeuroAnimator.

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

Table 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.

3.6 Example NeuroAnimators

We have successfully constructed and trained several NeuroAnimators to emulate a variety of physics-based models, including the 3-link pendulum from Fig. 5(a), a lunar lander spacecraft, a truck, and the dolphin model from Fig. 7. We used SD/FAST[4] to simulate the dynamics of the rigid body and articulated models, and we employ the simulator developed in [18] to simulate the deformable-body dynamics of the dolphin. Fig. 10 shows rendered stills from animations created using NeuroAnimators trained with these models.

Table 1 summarizes the structures of the NeuroAnimators developed to emulate these models (note that for the hierarchical dolphin NeuroAnimator, the table indicates the dimensions for only one of its six sub-networks; the other five 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 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. Section 5 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 of the associated physics-based models.

4 NeuroAnimator Controller Synthesis

We have demonstrated that it is possible to emulate a dynamical system using a trained neural network. We turn next to the problem of control; i.e., producing physically realistic animation that satisfies goals specified by the animator.

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

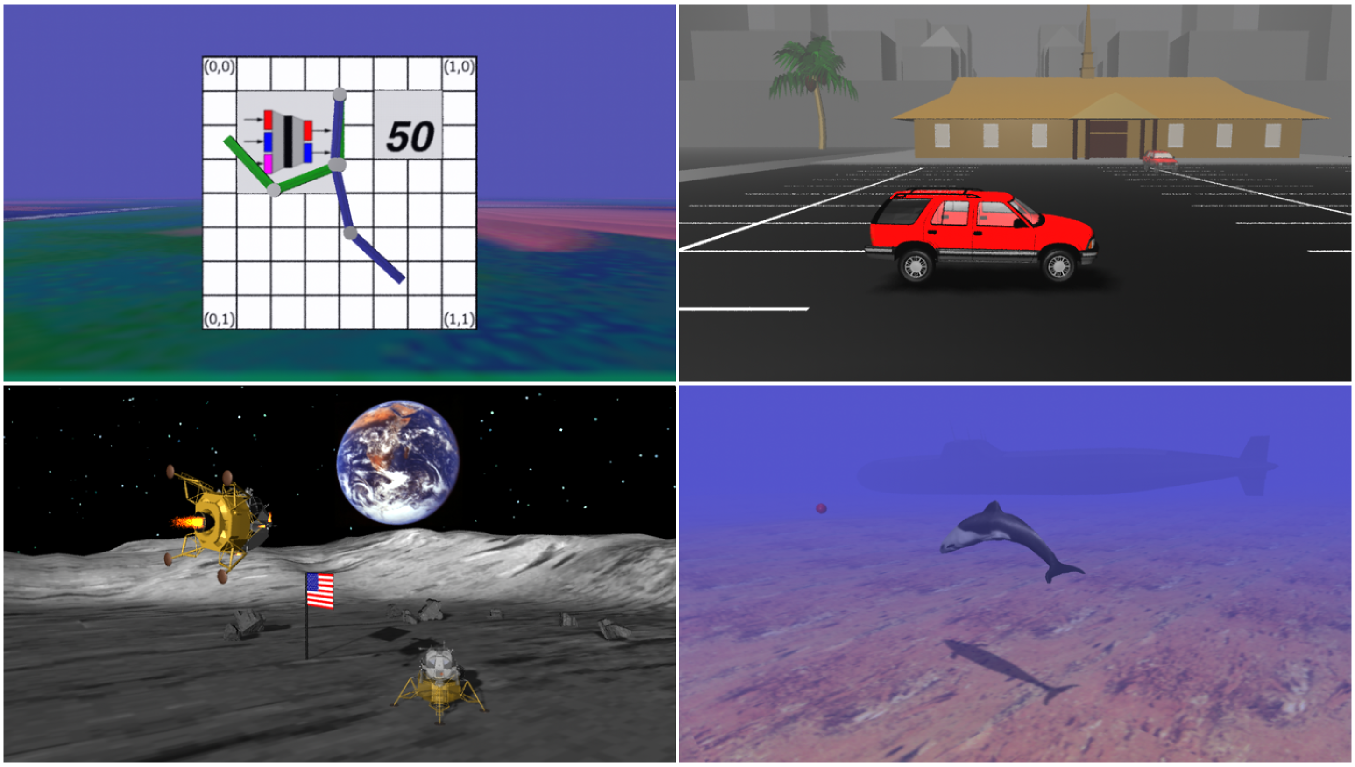


Figure 10: NeuroAnimators used in our experiments. The image at the upper left shows the emulator of a physics-based model of a planar multi-link pendulum suspended in gravity, subject to joint friction forces, external forces applied on the links, and controlled by independent motor torques at each of the three joints. The image at the upper right shows the emulator of 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. The image at the lower left shows the emulator of 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. The image at the lower right shows the emulator of a physics-based deformable (mass-spring-damper) model of a dolphin capable of locomoting via the coordinated contraction of 6 independently controlled muscle actuators which deform its body, producing hydrodynamic propulsion forces.

4.1 Motivation

A popular approach to the animation control problem is *controller synthesis* [11, 19, 5]. 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. Evaluation of the objective function requires a forward 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 [19, 5] and genetic algorithms [11]. 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 optimization methods guided by gradient directions.

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.

In the remainder of this section, 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.

4.2 Objective Function and Optimization

Using (9) we write a sequence of emulation steps

$$\mathbf{s}_{i+1} = \mathbf{N}_\Phi[\mathbf{s}_i, \mathbf{u}_i, \mathbf{f}_i]; \qquad 1 \le i \le M, \tag{10}$$

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. Figure 4 illustrates forward emulation by the NeuroAnimator according to this index notation.

Following the control learning formulation in [5], we define a discrete objective function

$$J(\mathbf{u}) = \mu_u J_u(\mathbf{u}) + \mu_s J_s(\mathbf{s}), \tag{11}$$

a weighted sum (with scalar weights μ_u and μ_s) of a term J_u that evaluates the controller $\mathbf{u} = [\mathbf{u}_1, \mathbf{u}_2, \dots, \mathbf{u}_M]$ 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 (10). 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

$$J(\mathbf{u}) = \frac{\mu_u}{2} \sum_{i=1}^{M} \mathbf{u}_i^2 + \frac{\mu_s}{2} (\mathbf{s}_{M+1} - \mathbf{s}_d)^2, \tag{12}$$

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.

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

$$\mathbf{u}^{l+1} = \mathbf{u}^l + \eta_x \nabla_{\mathbf{u}} J(\mathbf{u}^l), \tag{13}$$

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 (10) 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 $\mathbf{s}^l = [\mathbf{s}_1^l, \mathbf{s}_2^l, \ldots, \mathbf{s}_{M+1}^l]$, as is illustrated in Fig. 4. Next, it computes the components of $\nabla_{\mathbf{u}} J$ in (13) in an efficient manner. The cascade network structure enables us to apply the chain rule of differentiation within each network, chaining backwards across networks, yielding a variant of the backpropagation algorithm called *backpropagation through time* [15]. 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. 11 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 (13), 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 relearns very quickly.

More efficient optimization techniques can be applied to improve a slow convergence rate of the gradient descent algorithm (13). Adding momentum (7–8) to the gradient descent rule improves the effective learning:

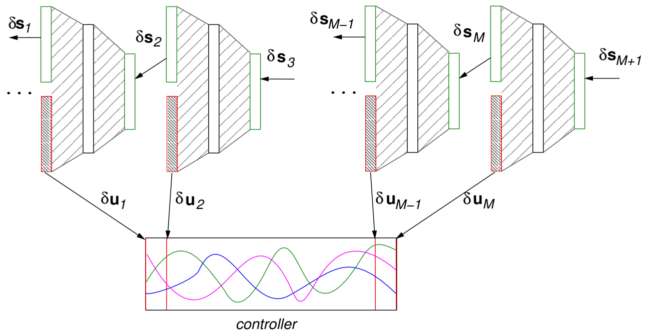


Figure 11: The backpropagation through time algorithm. At each iteration the algorithm computes the derivatives of the objective function with respect to the inputs of the emulator using the chain rule and it adjusts the control inputs to decrease the value of the objective function.

$$\delta\mathbf{u}^{l+1} = \eta_x \nabla_{\mathbf{u}} J(\mathbf{u}^l) + \alpha_x \delta\mathbf{u}^l, \tag{14}$$
$$\mathbf{u}^{l+1} = \mathbf{u}^l + \delta\mathbf{u}^{l+1}, \tag{15}$$

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. Section 6 includes a performance comparison of the different optimization techniques.

Up to now we have assumed that the objective is a known analytic function of the states and the controls of the model, as in (11). Although this definition covers a wide range of practical problems, our approach to control learning can handle objective functions whose analytic form is unknown in advance. See [4] for further discussion.

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.

5 NeuroAnimator Synthesis Results

As we discussed earlier, we have successfully constructed and trained several NeuroAnimators to emulate a variety of physics-based models pictured in Fig. 10. The ensuing discussion presents performance benchmarks and an error analysis.

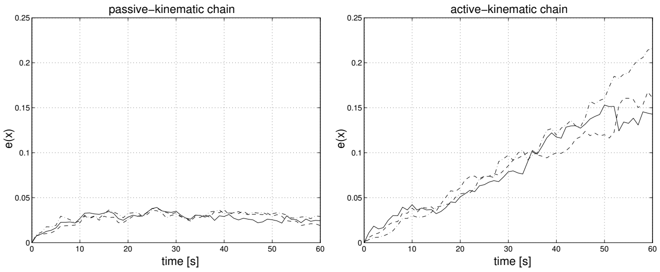
5.1 Performance Benchmarks

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 C 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. 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.

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 2 compares the physical simulation times obtained using the

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 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.



(a) (b)

Figure 12: 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. All experiments show the averaged error over 30 simulation trials and over all state variables. The duration of each trial was 6000 physical simulation timesteps.

SD/FAST physical simulator and 3 different neural network models: $\mathbf{N}_\Phi^{25}$, $\mathbf{N}_\Phi^{50}$, and $\mathbf{N}_\Phi^{100}$. For the truck model and the lunar lander model, we have trained only $\mathbf{N}_\Phi^{50}$ emulators. 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 physical model.

5.2 Approximation Error

As Fig. 12 shows, an interesting property of the neural network emulation is that the error does not increase appreciably for emulators with increasingly larger super 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 constant. This is attributable to the fact that an emulator that can predict further into the future must be iterated fewer steps per given interval of animation than an emulator that cannot 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, suggesting 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 the approximation error remained within reasonable bounds for the purposes of computer animation. 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 reproduces all of the visually salient properties of the physical motion.

5.3 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) emulation steps. Accumulated error can be annihilated by periodically performing regularization steps through the application of the true dynamical system (1) using an inexpensive, explicit Euler time-integration step

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

where the state is $\mathbf{s}_t = [\mathbf{v}_t^T, \mathbf{x}_t^T]^T$ and $\mathbf{d}(\mathbf{s}_t)$ are the deformation forces generated by the springs 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 inherent instability. To improve the stability when applying the explicit Euler step, we used a smaller spring stiffness and larger damping factor when compared to the semi-implicit Euler step used during the numerical simulation [18]. Otherwise the system would oscillate too much or would simply become unstable.

We achieve the best results when performing a few regularization steps after each emulation step. This produces much smoother motion than performing more regularization step but less frequently. Referring to Table 2 for the case of the deformable dolphin model, the second column indicates the simulation time using the physical simulator described in [18], the fourth column shows the simulation time using the $\mathbf{N}_\Phi^{50}$ emulator, and the last column reveals the impact of regularization on the emulation time. In this case, each emulation step includes 5 iterations of the above explicit Euler regularizer.

6 Control Learning Results

We have successfully applied our backpropagation through time controller learning algorithm to the NeuroAnimators presented in Section 5. 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. 13(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. 13(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. 13(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. 13(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.

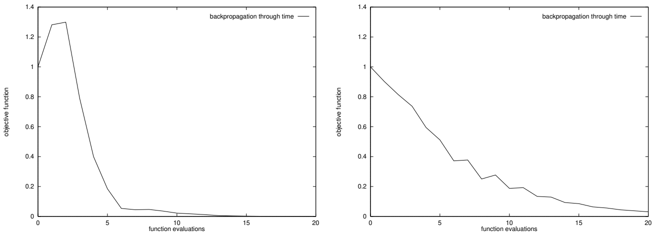
Referring to the locomotion learning problem studied in [5], we next compare the efficiency of our new backpropagation through time control learning algorithm using NeuroAnimators and the undirected search techniques—simulated annealing and simplex—reported in [5]. The locomotion learning problem requires the dolphin to learn how to actuate its 6 independent muscles over time in order to swim forward as efficiently as possible, as defined by an objective function that includes actuator work and distance traveled. Our earlier techniques take from 500 to 3500 learning iterations to converge because they need to perform extensive sampling of the control space in the absence of gradient information. By contrast, the gradient directed algorithm converges to a similar solution in as little as 20 learning iterations. Thus, the use of the neural network emulator offers a two orders of magnitude reduction in the number of iterations and a two orders of magnitude reduction in the execution time of each learning iteration. In terms of actual running times, the synthesis of the swimming controller which took more than 1 hour using the technique in [5] now takes less than 10 seconds on the same computer.

Hierarchically structured emulators, in which a global network represents the global aspects of motion and a set of sub-networks refine the motion produced by the global network, enable us to enhance the performance of our controller learning algorithm. For example, when applying the dolphin NeuroAnimator to learn locomotion controllers, we improve efficiency by employing only the global deformation network which accounts for the deformation of the entire body and suppressing the sub-networks that account for the local deformation of each body segment relative to its own center-of-mass coordinate system, since these small deformations do not significantly impact the locomotion. Similarly for a hierarchical human NeuroAnimator, when learning a controller that uses a subset of joints, we need only activate the sub-networks that represent the active joints.

We next compare the convergence of simple gradient descent and gradient descent with the momentum term on the control synthesis problem. Fig. 14 illustrates the progress of learning for the lunar lander problem. The results obtained using the momentum term are shown in the plot on the left and were generated using the parameters $\eta_x = 1.5$, $\alpha_x = 0.5$ in (14). The results obtained using the simple gradient descent are shown in the plot on the right and were generated using $\eta_x = 1.0$ in (13)—the largest learning rate that would converge. Clearly, the momentum term decreases the error much more rapidly, yielding an improved learning rate.

7 Conclusion

We have introduced the NeuroAnimator, an efficient alternative to the conventional approach of producing physically realistic animation through numerical simulation. NeuroAnimator involves the learning of neural network emulators of physics-based models by observing the dynamic state transitions produced by such models in action. The training takes place off-line and in advance. Animations subsequently produced by a trained NeuroAnimator approximate physical dynamics with dramatic efficiency, yet without serious loss of apparent fidelity. We have demonstrated the practicality of our technique by constructing NeuroAnimators for a variety of nontrivial physics-based models. Our unusual approach to physics-based animation furthermore led us to a novel controller synthesis method which exploits fast emulation and the differentiability of the NeuroAnimator approximation. We presented a "backpropagation through time" learning algorithm which computes controllers that satisfy nontrivial animation goals. Our new control learning algorithm is orders of magnitude faster than prior algorithms.



(a) (b)

Figure 14: The plots show the value of the objective as a function of the iteration of the control learning algorithm. The plot on the left was produced using the momentum term. It converges faster than simple gradient descent plotted on the right.

Acknowledgments

We 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. We are indebted to Steve Hunt for procuring the equipment that we needed to carry out our research at Intel. We thank Sonja Jeter for her assistance with the Viewpoint models and Mike Gendimenico for setting up the video editing suite and helping us to use it. We 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.

A Normalizing Network Inputs & Outputs

In Section 3.2 we recommended the normalization of emulator inputs and outputs. Variables in different groups (state, force, or control) require independent normalization. We normalize each variable so that it has zero mean and unit variance as follows:

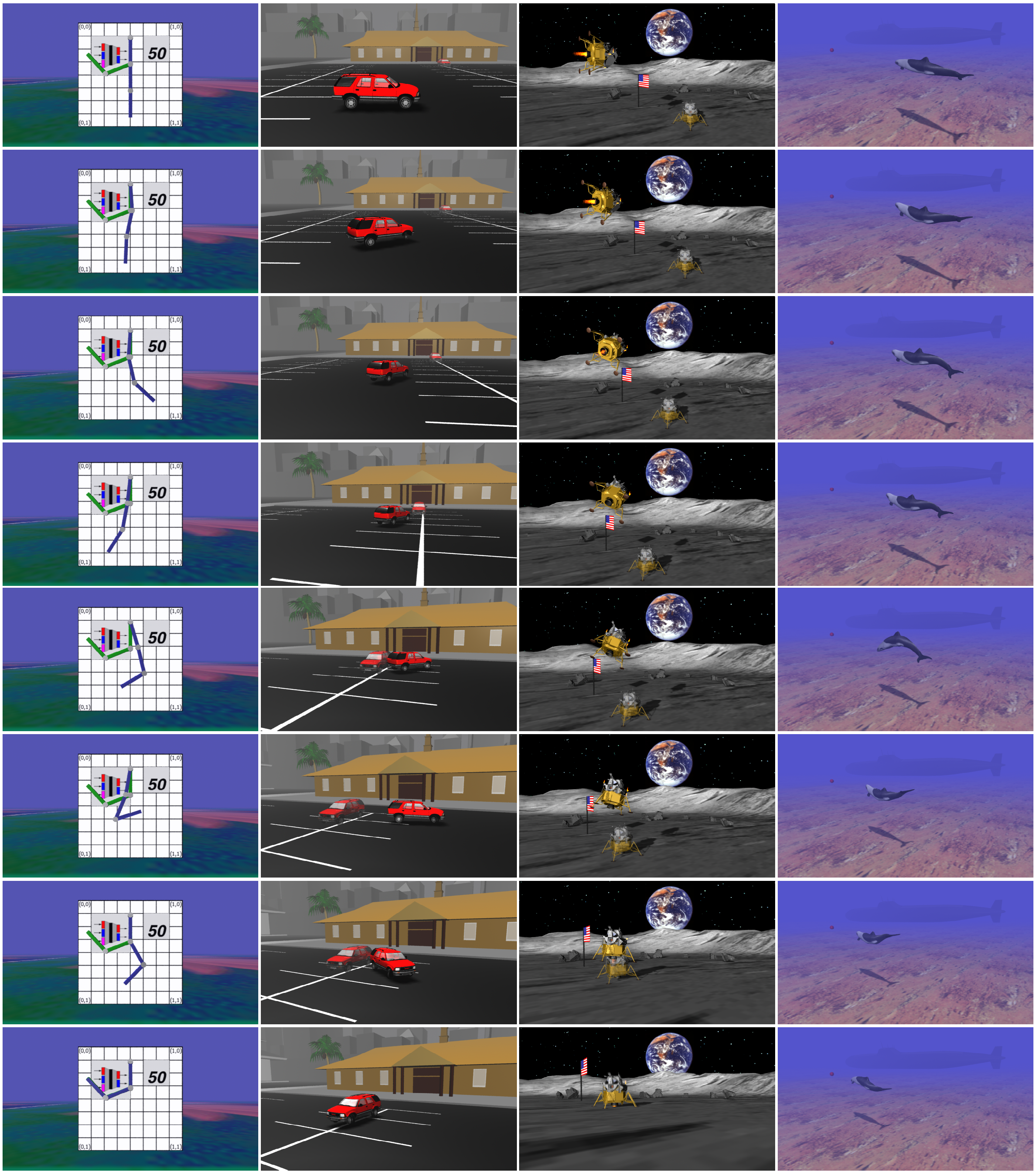
$$\tilde{x}_k^n = \frac{x_k^n - \mu_i^x}{\sigma_i^x}, \tag{16}$$

where the mean of the ith group of inputs is

$$\mu_i^x = \frac{1}{NK} \sum_{n=1}^{N} \sum_{k=k_i}^{k_i+K_i} x_k^n, \tag{17}$$

and its variance is

$$\sigma_i^x = \frac{1}{(N-1)(K-1)} \sum_{n=1}^{N} \sum_{k=k_i}^{k_i+K_i} (x_k^n - \mu_i)^2. \tag{18}$$



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

Figure 13: Results of applying the control learning algorithm to four different NeuroAnimators. (a) The 3-link pendulum NeuroAnimator in purple must reach the state indicated in green with zero final velocity. (b) The truck NeuroAnimator learning to park in the position and orientation of the translucent vehicle in the background. (c) The lunar lander NeuroAnimator learning to land with low descent velocity in the position and orientation of the translucent vehicle on the surface. (d) The dolphin NeuroAnimator learning to swim. The objective of locomoting as far forward as possible produces a natural, periodic swimming pattern.

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:

$$\tilde{y}_k^n = \frac{y_k^n - \mu_j^y}{\sigma_j^y}. \tag{19}$$

B Example Xerion Script

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.

```
#! /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
```

C 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 (k=0;k<inputSize;k++)
      hidden[j] += input[k]*ww[k][j];
    hidden[j]=hiddenLayer.transFunc(hidden[j]);
  }

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

References

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

[2] C. M. Bishop. *Neural Networks for Pattern Recognition*. Clarendon Press, 1995.

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

[4] R. Grzeszczuk. *NeuroAnimator: Fast Neural Network Emulation and Control of Physics-Based Models*. PhD thesis, Department of Computer Science, University of Toronto, May 1998.

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

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

[7] Jessica K. Hodgins, Wayne L. Wooten, David C. Brogan, and James F. O'Brien. Animating human athletics. In Robert Cook, editor, *SIGGRAPH 95 Conference Proceedings*, Annual Conference Series, pages 71–78. ACM SIGGRAPH, Addison Wesley, August 1995. held in Los Angeles, California, 06-11 August 1995.

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

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

[10] Gavin S. P. Miller. The motion dynamics of snakes and worms. In John Dill, editor, *Computer Graphics (SIGGRAPH '88 Proceedings)*, volume 22, pages 169–178, August 1988.

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

[12] D. Nguyen and B. Widrow. 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, 1989.

[13] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes: The Art of Scientific Computing, Second Edition*. Cambridge University Press, 1992.

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

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

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

[17] Demetri Terzopoulos, John Platt, Alan Barr, and Kurt Fleischer. Elastically deformable models. In Maureen C. Stone, editor, *Computer Graphics (SIGGRAPH '87 Proceedings)*, volume 21, pages 205–214, July 1987.

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

[19] Michiel van de Panne and Eugene Fiume. Sensor-actuator networks. In James T. Kajiya, editor, *Computer Graphics (SIGGRAPH '93 Proceedings)*, volume 27, pages 335–342, August 1993.