Seminararbeit

Fast Neural Network Emulation and Control of Physics-Based Models

Martin Spengler

15. Juni 1999

Zusammenfassung

Die vorliegende Arbeit entstand im Rahmen des Seminars *Aktuelle Themen der graphischen Datenverarbeitung* der *Computer Graphics Research Group* an der ETH Zürich.

Thema der Seminararbeit ist das Paper *NeuroAnimator: Fast Neural Network Emulation and Control of Physics-Based Models* [1] von Radek Grzeszczuk. Darin wird ein Verfahren vorgestellt, um Animationen von *physikbasierten Modellen* zu erzeugen. Das Verfahren, welches *künstliche neuronale Netzwerke* einsetzt, hat gegenüber dem klassischen Ansatz mittels Integration den Vorteil, numerisch um Grössenordnungen effizienter zu sein und trotzdem qualitativ gleichwertige Animationen zu erzeugen.

1 Einleitung

Animationen basierend auf den Prinzipien der Physik haben in den vergangenen Jahren zunehmend an Popularität gewonnen. Der Grund dafür liegt nicht einzig bei dem unübertroffenen Realismus der physikbasierten Animationstechniken. Es bestetht nämlich zusätzlich die Möglichkeit, durch Einführung passender Kontrollmechanismen eine Vielzahl qualitativ hochwertiger Animationen weitestgehend automatisch zu erzeugen. Dies steigert natürlich die Attraktivität der physikbasierten Animation und sorgt nicht zuletzt dafür, dass diese Techniken zunehmend in der kommerziellen Film- und Entertainmentbranche Einzug halten.

Als grösster Hemmschuh bei dieser Entwicklung erweist sich der gegenüber klassischen Animationsverfahren wie der geometrischen Modellierung immens gesteigerte Rechenaufwand. Genau dieses Problems der numerischen Kosten[1] nimmt sich der Ansatz von Grzeszczuk und Terzopoulos an. Der *NeuroAnimator* ermöglicht es, einen Emulator anhand von bestehenden Animationen zu trainieren, so dass er in der Lage ist, die gewünschten Animationen mit grösstmöglicher Äquivalenz zu reproduzieren. Allerdings verschlingt die Reproduktion der Animationssequenzen nur einen Bruchteil der Kosten, welche bei einer Neuberechnung der Animation anfallen würde.

Im Folgenden soll der Ansatz des *NeuroAnimators* erläutert werden: Abschnitt 2 führt die Konzepte der physikbasierten Animation ein, wie sie auch beim klassischen Ansatz der *numerischer Integration* zur Anwendung kommen. In Abschnitt 3 wird eine kurze Einführung in die Idee der *neuronalen Netzwerke* gegeben, welche für das Verständnis des im Abschnitt 4 präsentierten *NeuroAnimators* benötigt wird. Abschnitt 5 behandelt die Kontroller-Synthese mit neuronalen Netzwerken während in Abschnitt 6 abschliessend einige Reultate und Schlussfolgerungen präsentiert werden.

2 Physikalische Modelle

Unabhängig von der Modellbeschreibung (siehe Abschnitt 2.1) kann die Animation eines physikbasierten Modells als *numerische Simulation* der entsprechenden Bewegungsgleichungen gesehen werden. Dies führt zu der Berechnung eines zeitdiskreten dynamischen Systems der Form

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

Diese normalerweise *nichtlinearen* Gleichungen drücken den Vektor $\mathbf{s}_{t+\delta t}$ der Zustandsvariablen[2] des Systems zur Zeit $t + \delta t$ aus als Funktion $\mathbf{\Phi}$ des Zustandsvektors $\mathbf{s}_t$, des Kontrollvektors $\mathbf{u}_t$ und des

[1] Unter dem Ausdruck *Kosten* ist in diesem Zusammenhang der Aufwand an Rechenleistung bzw. Prozessorzeit zu verstehen.

[2] Freiheitsgrade des Systems und deren Geschwindigkeiten

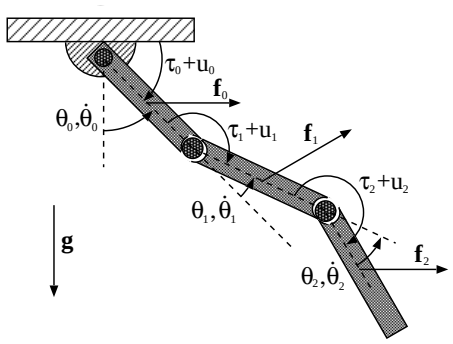

Abbildung 1: *Aktives* Multilink-Pendel: In den Gelenken können Momente appliziert werden.

Vektors $\mathbf{f}_t$ der externen Kräfte, welche zur Zeit t auf das System einwirken.

Die eigentliche Animation wird erzeugt, indem für jeden der $n+1$ Zeitschritte $t_i = t_0 + i \cdot \delta t$, $i = 0, \ldots, n$ die Funktion $\mathbf{\Phi}$ ausgewertet wird. Die Auswertung von $\mathbf{\Phi}$ ist in der Regel nichttrivial, kommen doch numerische Integrationsmethoden wie das Eulerverfahren oder der Runge-Kutta-Algorithmus zum Einsatz. Der Rechenaufwand bewegt sich dabei im Bereich von $O(N)$ für explizite Integrationsverfahren bis $O(N^3)$ für implizite Verfahren, wobei N proportional zur Dimension des Zustandsraumes ist.

2.1 Modelle

Wie muss nun aber ein Modell beschrieben werden, damit es physikalisch korrekt animiert werden kann? – Grundsätzlich gibt es nur eine einzige Vorgabe bzw. Einschränkung: Die beschreibenden Gleichungen müssen den Gesetzen der Physik gehorchen.

Trotz dieser theoretisch grossen Freiheit in der Wahl der Modellbeschreibung finden in der Praxis zwei Modellvorstellungen Verwendung: Die Beschreibung eines Objekts als System von *starren Körpern* (rigid bodies), welche durch Gelenke verbunden sind und die Modellierung als *Feder-Masse-System* (mass-spring system).

Das Multilink-Pendel aus Abb. 1 ist ein klassisches Beispiel für die Modellierung mit starren Körpern. Es besteht eine offensichtliche Verwandschaft dieses Beispiels zu den Manipulatoren[3], welche bei Industrierobotern zu finden sind. Es ist daher wenig überraschend, dass bei der Modellierung des Pendels dieselben mathematischen und physikalischen

[3] Dies gilt für *aktive* Pendel, bei denen gesteuerte Motorkräfte bzw. Drehmomente an den Gelenken einwirken.

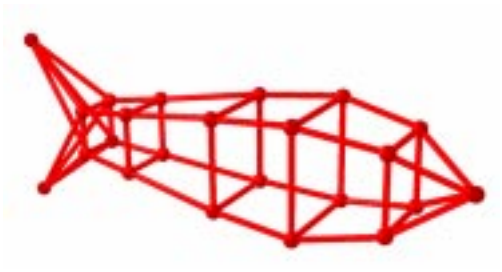

Abbildung 2: Feder-Masse-Skelett eines Fischmodells

Werkzeuge zum Einsatz kommen, wie sie von der Robotik zur Beschreibung der *Kinematik* von Robotermanipulatoren hervorgebracht wurden.

Weiter eignet sich das Konzept der starren Körper zur Modellierung von Fahr- und Flugzeugen bzw. von tierischen und menschlichen Körpern. Allerdings wird bei der Modellierung organischer Körper prinzipbedingt deren Verformbarkeit aufgegeben.

Wird diese Flexibilität oder Verformbarkeit verlangt, bietet sich das zweite Modellierungskonzept an: Die Modellierung eines Objektes als *Feder-Masse-System*. Dabei wird die Modellvorlage als Menge von (Punkt-)Massen beschrieben, welche über Federn mit spezifischen Eigenschaften wie Ruhelänge, Federkonstante und Dämpfungskoeffizient verbunden sind.

Ein beeindruckendes Beispiel für ein solches Modell aus dem Reich der Unterwassertierwelt findet sich in [3]: Es wird ein Fischmodell (Abb. 2) präsentiert, welches nicht nur in der Lage ist, realistische Schwimmbewegungen auszuführen, sondern auch ein komplett verhaltengesteuertes (*behaviour based*) Eigenleben entwickelt. Der eigentliche Fischkörper wird dabei durch NURBS[4] modelliert. Diese "Aussenhaut" wird an einem Feder-Masse-Skelett verankert, welches der eigentlichen Animation dient. Das Animationsmodell ist durch ein System von *Lagrange'schen Bewegungsgleichungen* beschrieben:

$$m_i \frac{d^2\mathbf{x}_i}{dt^2} + \rho_i \frac{d\mathbf{x}_i}{dt} - \mathbf{w}_i = \mathbf{f}_i^w \qquad (2)$$

Hierbei bezeichnet m_i die Masse des Knotens i, welcher sich an der Position $\mathbf{x}_i(t) = [x_i(t), y_i(t), z_i(t)]$ befindet. Seine Geschwindigkeit und Beschleunigung ergibt sich zu $\mathbf{v}_i(t) = d\mathbf{x}_i/dt$ bzw. $\mathbf{a}_i(t) = d^2\mathbf{x}_i/dt^2$. Die elastische Feder S_{ij} verbindet die beiden Knoten i und j, wobei c_{ij} die entsprechende Federkonstante und l_{ij} die Auslenkung von S_{ij} in

[4] NURBS: *Non Uniform Rational B-Splines*

Ruhelage ist. Der Dämpfungsfaktor wird mit ρ_i bezeichnet. Der Knoten i erfährt von jedem seiner Nachbarknoten $j \in N_i$ (N_i =Menge der Nachbarknoten von i) eine Kraft $\mathbf{f}_{ij}^s(\mathbf{x}_i, \mathbf{x}_j)$. Die Resultierende der auf einen Knoten i einwirkenden internen Kräfte ist $\mathbf{w}_i(t) = \sum_{j \in N_i} \mathbf{f}_{ij}^s$. In (2) entspricht $\mathbf{f}_i^w(t)$ der auf den Knoten i einwirkende externen Kraft. Im Falle des Fischmodelles ist dies eine hydrodynamische Kraft, welche die Vorwärtsbewegung des Fisches bewirkt.

2.2 Kontroller

Bis jetzt sind die vorgängig vorgestellten Modelle *passiv*, d.h. sie sind einzig dem Spiel allfälliger Kräfte ausgesetzt, die von der Umwelt auf das Modell einwirken. Damit interessante Animationen erzeugt werden können, muss ein Mechanismus gegeben sein, um auf das Verhalten des Modells Einfluss zu nehmen. In dem System (1) wird dieser Steuerungsmechanismus mit dem *Kontrollvektor* $\mathbf{u}_t$ implementiert.

Durch den Vektor $\mathbf{u}_t$ können Steueranweisungen für die Berechnung des nächsten Animationsschrittes $\mathbf{s}_{t+\delta t}$ an die Funktion $\mathbf{\Phi}$ übergeben werden. Für das aktive Multilink-Pendel in Abb. 1 sind das die von Motoren erzeugten Drehmomente u_0 bis u_2, welche der Gewichtskraft $\mathbf{g}$ und den Reibungsverlusten τ_0, τ_1 und τ_2 in den Gelenken entgegen wirken. Die Animation des Pendels entspricht also der Generierung einer zeitlichen Abfolge von Motormomenten, welche das Pendel die gewünschten Bewegungen ausführen lassen[5].

Das Fischmodell aus Abb. 2 hingegen benötigt eine periodische Folge von "Muskelkontraktionen"[6] um eine Vorwärtsbewegung zu erzeugen.

Prozeduren bzw. Algorithmen, die solche Kontrollvektoren $\mathbf{u}_t$ erzeugen, werden *Kontroller* genannt. Sie haben die Funktion einer Abbildung zwischen zwei unterschiedlich hohen Abstraktionsleveln. Beispielsweise könnte für das Fischmodell ein Kontroller bestehen, der eine Sequenz von Vektoren $\mathbf{u}_t$ erzeugt, um den Fisch mit Geschwindigkeit v geradeaus schwimmen zu lassen: $forward(\mathbf{s}_t, v) \mapsto \mathbf{u}_{t+\delta t}$.

Stellt sich noch die Frage, wie solche Kontroller erzeugt werden. In der Praxis haben sich zwei Verfahren als mehr oder weniger parktikabel erwiesen:

"Handarbeit" Gemeint ist, dass der Kontroll-Algorithmus "von hand" entworfen und entwickelt wird. Mit anderen Worten: Die gesuchte Abbildung wird irgendwie aus Messungen und Beobachtungen an der realen Modellvorlage extrahiert. Für das Fischmodell wird also z.B. das Schwimmverhalten von richtigen Fischen studiert (siehe [3]). Damit dieser Ansatz erfolgreich sein kann, wird neben einer ordentlichen Portion Intuition und Fachwissen aus dem Bereich des zu animierenden Objekts ein künstlerisches Flair benötigt.

Adaptive Lernverfahren Im Gegensatz zur Methode der "Handarbeit", deren Erfolg in erster Linie von dem Können des Implementators abhängt, bieten die adaptiven Lernverfahren die Möglichkeit, die gesuchten Kontroller mehr oder weniger automatisch zu erzeugen. Dazu werden wohlbekannte Lernverfahren wie *(un)supervised learning* oder *genetische Algorithmen* eingesetzt. Diese Techniken aus den Gebieten *Funktionsoptimierung*, *computational intelligence* und *artificial life* versuchen, das Verhalten des Modells hinsichtlich einer gegebenen Bewertungsfunktion zu optimieren, was zu ganz erstaunlichen Ergebnissen und hochkomplexen Bewegungsmustern führen kann.

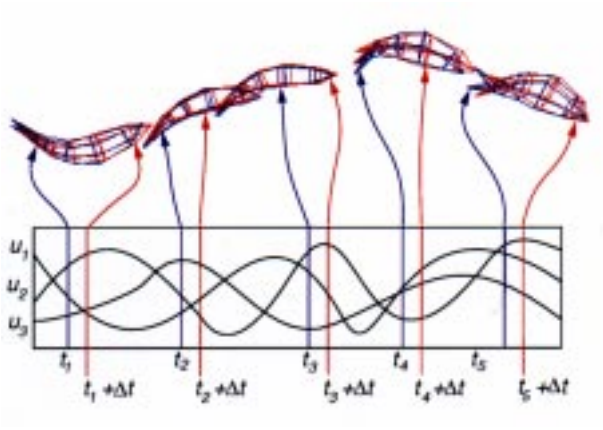

Abbildung 3: Kontroller für die Schwimmbewegung eines Delphin-Modells.

Wie adaptive Lernverfahren zur Kontrollersynthese eingesetzt werden, wird in Abschnitt 5 näher erläutert.

3 Neuronale Netze

Bevor in Abschnitt 4 der eigentliche *NeuroAnimator* betrachtet wird, werden in dem folgenden Abschnitt

[5] Vergleiche mit der aus der Robotik bekannten *inversen Kinematik*.

[6] Rythmisches Verkürzen bzw. Verlängern der Ruhelänge l_{ij} bestimmter, als "Muskeln" bezeichneter Federn S_{ij}

die Grundideen und -konzepte der *künstlichen* neuronalen Netzwerke[7] eingeführt. Aus Gründen der Zweckmässigkeit beschränken wir uns dabei auf den verbreiteten Typ des *Multilayer-Perzeptrons*[8]. Eine allgemeine Einführung in die umfangreiche Thematik der neuronalen Netzwerke bietet z.B. [4].

Aus mathematischer Sicht sind neuronale Netzwerke komplexe *Abbildungen* der Form $\Re^p \mapsto \Re^r$, wobei p die Anzahl Eingänge und r die Anzahl Ausgänge des Netzwerks bezeichnen. Implementiert wird eine solche Abbildung als Netzwerk von vielen, vergleichsweise einfachen Zellen, die über veränderbare Verbindungen interagieren. Die zweite wichtige Eigenschaft neuronaler Netzwerke ist die *Lernfähigkeit*, welche in der Veränderbarkeit der Zellverbindungen begründet ist. In wieweit diese Eigenschaften für die Animation physikbasierter Modelle von Bedeutung sind, erläutert Abschnitt 4.

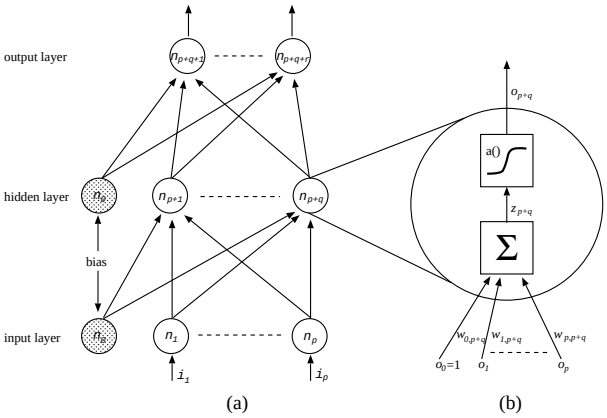

Abbildung 4: (a) Biased Multilayer-Perzeptron wie er im NeuroAnimator verwendet wird. (b) Mathematisches Modell eines Neurons.

3.1 Der Perzeptron

Wie eingangs erwähnt, soll im Rahmen dieser Arbeit nur der Multilayer-Perzeptron bzw. um präzise zu sein, der *biased Multilayer-Perzeptron* betrachtet werden. Abb. 3.1 zeigt ein Perzeptron der Art, wie er im NeuroAnimator zum Einsatz kommt. Im weiteren soll, wenn nicht anders angegeben, der Begriff "Perzeptron" gleichbedeutend mit "biased Mulitlayer-Perzeptron" sein.

Ein Perzeptron besteht aus einer Anzahl *Neuronen*, welche in Schichten (*layers*) organisiert und untereinander verbunden sind. Dabei kann ein Neuron der Schicht l nur Signale von Neuronen der Schicht $l-1$ empfangen. Daher bezeichnet man den Perzeptron auch als *feedforward* Netzwerk. Es gibt drei Sorten von Neuronenschichten: Die Eingabeschicht (*input layer*), die Ausgabeschicht (*output layer*) und die Zwischenschichten (*hidden layers*)[9].

Das zentrale Element der neuronalen Netzwerke, das *Neuron*, ist wie folgt aufgebaut: In einem ersten Schritt werden die Ausgangssignale o_i der p Vorgängerneuronen (predecessor) $i \in P_j$ von Neuron j mit den entsprechenden Verbindungsgewichten w_{ij} skaliert und zum Signal z_j aufsummiert. Dies ist vergleichbar mit der Funktion der *Dendriten* und *Synapsen*, wie sie im biologischen Neuron vorkommen.

$$z_j = w_{0j} + \sum_{i=1}^{p} o_i w_{ij} = \sum_{i=0}^{p} o_i w_{ij} = \mathbf{x}^T \mathbf{w}_j \qquad (3)$$

Weiter wird zum Signal z_j der *Bias* w_{0j} hinzu addiert. Ihm kommt die Bedeutung eines Schwellenwertes zu. Fügt man der Menge P_j der Vorgängerneuronen von Neuron j ein Neuron n_0 mit konstantem Ausgangssignal $o_0 = 1$ hinzu, kann auf eine gesonderte Behandlung des Bias-Parameters w_{0j} verzichtet werden.

In einem nächsten Schritt wird das Ausgabesignal o_j des Neurons j berechnet. Dazu wird das dendritische Signal z_j an eine stetige, monotone, oft nichtlineare *Aktivierungsfunktion* $\mathrm{a}(z)$ übergeben: $o_j = \mathrm{a}(z_j)$. Üblicherweise wird für die Aktivierungsfunktion die *logistische Funktion* $\mathrm{a}(z) = \frac{1}{1+e^{-z}}$ oder aber der *Tangens hyperbolicus* $\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$ verwendet, da diese einen sigmoiden Funktionsverlauf und somit eine Art Schwellenverhalten aufweisen.

3.2 Lernen in Perzeptrons

Nachdem der Aufbau und die Funktionsweise des Perzeptrons erläutert wurde, stellt sich die Frage, wie man die Gewichte w_{ij} wählen muss, um ein gewünschtes Verhalten des Netzwerkes zu erzielen.

Angenommen, wir haben einen zweischichtigen Perzeptron[10] mit p Eingabeneuronen, q Neuronen

[7] Im Gegensatz dazu stehen die *biologischen* neuronalen Netzwerke, wie wir sie z.B. im menschlichen Gehirn finden.

[8] Fälschlicherweise werden die Begriffe *neuronales Netzwerk* und *Perzeptron* oft gleichgesetzt. Tatsache ist aber, dass der Perzeptron nur ein Typ unter vielen ist. Für eine breitere Übersicht sei an dieser Stelle auf [4] verwiesen.

[9] Im Gegensatz zum *Multilayer-Perzeptron*, welcher mindestens eine Zwischenschicht aufweist, besteht ein *Perzeptron* wie ihn Rosenblatt ursprünglich beschrieben hat, nur aus einer Eingabeschicht und einer Ausgabeschicht.

[10] Das heisst, ein Netzwerk mit *einem* hidden layer. Es

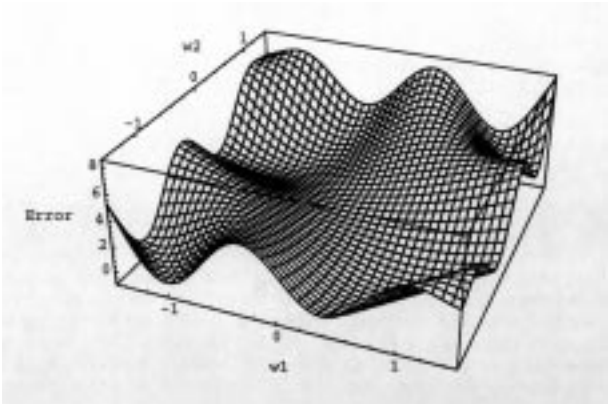

Abbildung 5: Fehlerfläche eines Netzwerkes mit zwei Eingabe- und einem Ausgabeneuron.

in der Zwischenschicht, r Ausgabeneuronen und dem Gewichtsvektor $\mathbf{w}$. Dann definiert der Perzeptron $\mathbf{N}(\mathbf{x}, \mathbf{w})$ eine stetige Abbildung $\mathbf{N} : \Re^p \mapsto \Re^r$. Ausgehend von einem ausreichend grossen q, wird ein feedforward Netzwerk der beschriebenen Art eine stetige Abbildung $\Phi : \Re^p \mapsto \Re^r$ über einen kompakten Bereich $\mathbf{x} \in \chi$ mit beliebiger Genauigkeit approximieren [2].

Mit anderen Worten existiert für ein ausreichend kleines $\epsilon > 0$ ein Netzwerk $\mathbf{N}$, so dass gilt

$$\forall \mathbf{x} \in \chi, \qquad e(\mathbf{x}, \mathbf{w}) = \|\Phi(\mathbf{x}) - \mathbf{N}(\mathbf{x}, \mathbf{w})\|^2 < \epsilon, \quad (4)$$

mit dem *Approximationsfehler* e.

Ein neuronales Netzwerk kann die Approximation einer Abbildung Φ *lernen*. Der Lernprozess in einem neuronalen Netzwerk ist also nichts anderes, als eine Fehlerminimierung. In Abb. 5 ist die Fehlerfunktion $e(\mathbf{x}, \mathbf{w})$ für ein Netzwerk mit zwei Eingabe- und einem Ausgabeneuron dargestellt. Ein Lernalgorithmus hat nun zum Ziel, ein globales oder wenigstens lokales Minimum dieser Fehlerfläche zu finden.

Die bekannten Lernalgorithmen lassen sich grob in drei Kategorien unterteilen:

supervised learning Beim supervised learning werden dem Netzwerk eine Reihe von *Trainingsdaten* präsentiert. Diese bestehen aus Ein-/Ausgabe-Paaren, welche die zu approximierende Abbildung Φ möglichst gut sampeln. Aus der Differenz zwischen errechneter und gefordeter Ausgabe lässt sich anschliessend eine Korrektur der Verbindungsgewichte w_{ij} berechnen.

reinforcment learning Auch hier werden dem Neztwerk Trainingsdaten präsentiert, allerdings diesmal nur die Eingabedaten. Ein "Trainer" bewertet die Ausgabe des Netzwerkes und dieses berechnet davon ausgehend eine allfällige Korrektur der Verbindungsgewichte.

unsupervised learning Dem Netzwerk werden wiederum nur Eingabedaten als Trainingsmuster präsentiert. Das Netzwerk versucht dann, sich entsprechend selber zu organisieren bzw. zu trainieren. Dieses Lernverfahren findet vor allem bei den sog. *Kohonen-Karten* oder *self-organizing maps* (SOM) Anwendung.

Im weiteren soll näher auf das *supervised learning* eingegangen werden, da dieses im *NeuroAnimator* in Form des *Backpropagation-Algorithmus'* Anwendung findet.

Wie bereits angesprochen, besteht ein Trainingsbeispiel für das *supervised learning* aus einem Paar von Ein- und Ausgabesignalen:

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

Hierbei ist $\mathbf{x}^\tau$ der Eingabevektor und $\mathbf{y}^\tau$ der gewünschte Ausgabevektor. Der Lernalgorithmus sucht nun eine Menge von Verbindungsgewichten $\mathbf{w}$ für das Netzwerk $\mathbf{N}(\mathbf{x}, \mathbf{w})$, so dass die Differenz zwischen der berechneten und der gewünschten Ausgabe ausreichend klein ist bzw. Gl. 4 genügt.

3.3 Backpropagation

Ein verbreiteter Lernalgorithmus aus der Kategorie des *supervised learning* ist der *Backpropagation-Algorithmus*. Es handelt sich dabei um ein Gradientenabstigesverfahren wie es etwa *steepest descent* auch ist.

Der Backpropagation-Algorithmus versucht die Fehlerfunktion

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

zu minimieren, welche den Approximationsfehler e aus Gl. 4 über alle n Trainingsbeispiele aufsummiert. Der *off-line*-Algorithmus justiert die Verbindungsgewichte des Netzwerkes gemäss der Formel

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

werden nämlich sinnvollerweise die Anzahl der Verbindungsschichten gezählt, da in diesen die Veränderungen bzw. der Lernprozess stattfinden!

wobei $\nabla_{\mathbf{w}} E$ den Gradienten der Fehlerfunktion bezüglich der Gewichte $\mathbf{w}$ bezeichnet und $\eta < 1$ die *Lernrate* genannt wird.

Die *on-line*-Variante des Backpropagation-Algorithmus korrigiert die Verbindungsgewichte nach jedem präsentierten Trainingsbeispiel. Näheres dazu findet sich in [4].

Beim Backpropagation-Algorithmus handelt es sich um eine rekursiven Algorithmus, welcher sich ausgehend von den Ausgabeneuronen rückwärts zu den Eingabeneuronen durcharbeitet und die Verbindungsgewichte zu optimieren versucht.

4 NeuroAnimator

Nachdem in den vorangehenden drei Abschnitten vor allem Grundlagen vermittelt wurden, soll in den restlichen Abschnitten der sog. *NeuroAnimator* von Grzeszczuk und Terzopoulos präsentiert und besprochen werden.

Der NeuroAnimator entstand aus dem Verlangen nach einer Methode, die physikbasierende Animationen kostengünstig emulieren kann. Mit anderen Worten: Es wurde ein Verfahren gesucht, welches qualitativ hochwertige Animationen erzeugt, ohne Resourcen in der Grössenordung der traditionellen Animationsverfahren zu verschlingen.

Neben dieser primären Aufgabe kann der NeuroAnimator weiter zur Synthese von Kontrollern, wie sie in Abschnitt 2.2 eingeführt wurden, verwendet werden.

4.1 Animation

Ausgehend von dem System (1), muss ein neuronales Netzwerk konstruiert werden, welches eine möglichst gute Approximation von Φ in (1) liefert. Wie in Abschnitt 3.1 bereits angekündigt,verwenden wir dazu ein *biased Multilayer-Perzeptron* $\mathbf{N}_\Phi$, welcher mit dem *Backpropagation*-Lernalgorithmus trainiert wird.

Das günstige Verhalten solcher Perzeptrons erlaubt es sogar, die im System (1) verwendete Samplingrate δt der Animation durch eine Art *Super-Zeitschritt* $\Delta t = n\delta t$ zu ersetzen. Dadurch müssen weniger eigentliche Emulationsschritte berechnet werden, was zusätzlich Zeit und Aufwand einspart. Analog zum System (1) kann also ein Emulationsschritt folgendermassen beschrieben werden:

$$\mathbf{s}_{t+\Delta t} = \mathbf{N}_\Phi[\mathbf{s}_t, \mathbf{u}_t, \mathbf{f}_t]. \tag{7}$$

Da ein Emulationsschritt Δt im Vergleich zu einem Schritt δt des physikalischen Simulation gross ist, können bei einer emulierten Animation unschöne Artefakte auftreten. Um diese zu vermeiden, wird die Bewegung des Modells in der Framerate der Animation gesampelt und die Frames zwischen den emulierten Frames des NeuroAnimators *linear interpoliert*. Nach [1] genügt eine lineare Interpolation, um flüssige Animationen zu erzeugen. In der Anwendung höherwertiger Interpolationsverfahren liegt aber noch Potential, um die Qualität der erzeugten Animationen zu verbessern.

4.2 Emulation

Der *NeuroAnimator* ist als zweischichtiger Perzeptron (*2-layer perceptron*) aufgebaut. Die Anzahl der Neuronen in der Zwischenschicht (*hidden neurons*) ist dabei von der jeweiligen Aufgabenstellung abhängig[11]. Als Aktivierungsfunktion der *hidden neurons* wird die logistische Funktion verwendet, wie sie in Abschnitt 3.1 beschrieben ist.

Ein trainiertes Netzwerk $\mathbf{N}_\Phi$ erhält als Eingabe einerseits den Vektor $\mathbf{s}_t$, der den Zustand des Modells zum Zeitpunkt t wiederspiegelt, und andererseits eine beliebige Kombination aus den Vektoren $\mathbf{u}_t$ und $\mathbf{f}_t$. Beliebig heisst an dieser Stelle, dass entweder beide Vektoren dem Netzwerk als Input zugeführt werden oder aber nur einer bzw. gar keiner verwendet wird.

Die Situation, in der neben $\mathbf{s}_t$ nur noch der Vektor der externen Kräfte $\mathbf{f}_t$ auftritt, findet man bei *passiven* Modellen. Die Abwesenheit von $\mathbf{f}_t$ hingegen weist darauf hin, dass die externen Kräfte komplet durch den Status des Systems $\mathbf{s}_t$ determiniert sind.

Als Ausgabe liefert das Netzwerk einen Statusvektor $\mathbf{s}_{t+\Delta t}$, welcher beim nächsten Emulationsschritt wiederum als Eingabe dient (Abb. 6). Eigentlich ist das Netzwerk also *rekursiv*.

4.3 Training

Um ein physikbasiertes Modell mit einem NeuroAnimator zu emulieren, muss das zugrundeligende neuronale Netzwerk erst trainiert werden. Es wird dazu eine Menge von Trainingsdaten der Art $\mathbf{x}^\tau, \mathbf{y}^\tau, \tau = 1, 2, \ldots$ durch die numerische Simulation des Modells erzeugt. Basierend auf diesen Datenpaaren justiert der Backpropagation-Algorithmus die Verbindungsgewichte des NeuroAnimators.

Offensichtlich ist ein solches Training sehr langwierig und zeitintensiv, macht es doch die Generierung und Verarbeitung einer grossen Anzahl von Trainingsdaten nötig. Ist der NeuroAnimator aber

[11] übliche Zahlen sind 20 bis 50 Neuronen in der Zwischenschicht.

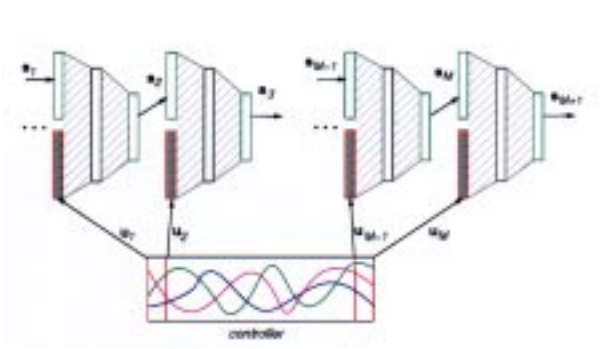

Abbildung 6: Emulation mit neuronalem Netzwerk. Bei jeder Iteration wird der vorhergehende Output zum aktuellen Input. Der Algorithmus ist demnach *rekursiv*.

einmal trainiert, kann er immer wieder verwendet werden um Animationen des Modells ohne grossen Rechenaufwand, d.h. schnell zu erzeugen. Weiter ist er oft auch in der Lage, Bewegungsabläufe zu generieren, welche er nie zuvor erlernt hat.

Wie aber trifft man eine sinnvolle Auswahl der verwendeten Trainingsdaten? — Ziel des Trainings ist es ja, einen Emulator zu erzeugen, der alle nur erdenklichen Kontrollinputs $\mathbf{u}_{\Delta t}$ in die entsprechenden Bewegungen des emulierten Modells umsetzt. Dazu werden eine Reihe von numerischen Simulationen des betrachteten Modells gerechnet, jeweils mit zufällig gewählten Initialwerten $\mathbf{s}_0, \mathbf{u}_0$ und $\mathbf{f}_0$. Während der Simulation werden dann Trainingsbeispiele der Art

$$\{[\mathbf{s}_{t_k}^T, \mathbf{u}_{t_k}^T, \mathbf{f}_{t_k}^T]^T; \mathbf{s}_{t_k+\Delta t} \qquad k = 1, 2, \ldots\} \tag{8}$$

aufgezeichnet. Dabei wird darauf geachtet, dass die einzelnen Trainingspaare einer Simulation zeitlich ausreichend weit von einander entfernt sind: $\Delta t \leq (t_{k+1} - t_k) \leq 5\Delta t$. Dies verhindert eine übermässige Korrelation zwischen den Trainingsdaten, was sich seinerseits positiv auf das Lernverhalten des neuronalen Netzwerks auswirkt. Zusätzlich wird die Reihenfolge, in der die Trainingsbeispiele dem Backpropagation-Algorithmus präsentiert werden *randomisiert*. Im Gegensatz zu einem *motion tracking*-Verfahren, sieht ein NeuroAnimator während seiner Trainingsphase nie komplete Trajektorien der beobachteten Modelle, sondern nur unabhängige Einzelbeispiele in zufälliger Folge.

5 Neuro-Kontroller

Neben der eigentlichen Emulation von Animationen physikbasierter Modelle, lässt sich der NeuroAnimator auch zur sog. Kontroller-Synthese (*controller synthesis*) verwenden. Hierbei geht es also um die Erschaffung eines Kontrollers, der physikalisch realistische Animationen erzeugt, die die vom Animator spezifizierten Ziele erfüllen (vgl. Abschnitt 2.2).

5.1 Kontroller-Synthese

Bei den klassischen Ansätzen zur Kontroller-Synthese wird üblicherweise eine *trial and error*-Strategie eingesetzt: Durch wiederholtes numerisches Simulieren des physikbasierten Modells wird versucht, den Kontroller zu optimieren. Dazu bewertet eine Zielfunktion, inwieweit die durch den Kontroller erzeugte Animation des Modells die tatsächliche Zielsetzung erfüllt. Nach jedem Simulationsdurchgang wird diese Zielfunktion evaluiert und der Kontroller optimiert.

Diese Zielfunktion ist leider — durch den hohen Komplexitätsgrad des Optimierungsproblems bedingt — selten eine analytisch differenzierbare Funktion. Deshalb sind Gradientenabstiegs-Verfahren zur Optimierung von vornherein von der Anwendung ausgeschlossen und es bleiben noch Methoden wie *simulated annealing* oder *genetische Algorithmen*.

Die genannten Klassen von Algorithmen basieren auf einem *random walk* durch den gesamten Suchraum aller möglichen Kontroller. — Eine Vorgehensweise, welche im Vergleich zu den Gradienten-Abstiegs-Verfahren sehr langsam konvergiert.

Dem gegenüber bietet der NeuroAnimator den weiteren Vorteil, dass er es erlaubt, den gesuchten Gradienten zu berechnen und desshalb auch auf die viel effizientere Klasse der Gradientenabstiegs-Algorithmen zur Kontroller-Optimierung zurückgegriffen werden kann. Es wird daher in den folgenden Abschnitten ein Verfahren umrissen, welches die Kontroller-Synthese basierend auf dem NeuroAnimator ermöglicht.

5.2 Zielfunktion und Training

Als Zielfunktion für die Kontroller-Synthese mit dem NeuroAnimator erwies sich eine gewichtete Summe der Art

$$J(\mathbf{u}) = \mu_u J_u(\mathbf{u}) + \mu_s J_s(\mathbf{s}) \tag{9}$$

als geeignt, wobei μ_u und μ_s skalare Gewichte sind. Die Summe selber besteht im Wesent-

lichen aus einem Term J_u, der den Kontroller $\mathbf{u} = [\mathbf{u}_1, \mathbf{u}_2, \ldots, \mathbf{u}_M]$ evaluiert und einem Term J_s, welcher dasselbe für den Systemzustand $\mathbf{s} = [\mathbf{s}_1, \mathbf{s}_2, \ldots, \mathbf{s}_{M+1}]$ macht. Die beiden Sequenzen $\mathbf{u}$ und $\mathbf{s}$ stehen über die Relation

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

miteinander in Verbindung. Mit anderen Worten: Die Zustandssequenz $\mathbf{s}$ wird durch den NeuroAnimator unter Berücksichtigung der Kontroller-Sequenz $\mathbf{u}$ erzeugt.

Die Bewertung der Zielfunktionen erfolgt entsprechend den gemachten Vorgaben in den beiden Evaluations-Termen J_u und J_s. Beispielsweise kann der Kontroller über J_u dahingehend optimiert werden, dass die erzeugten Kontrollsequenzen energetisch minimal sind. Über J_s lassen sich demgegenüber Optimierungskriterien hinsichtlich der verfolgten Trajektorie des Modells implementieren.

Folgende Zielfunktion wurde von Grzeszczuk und Terzopoulos in ihren Experimenten erfolgreich verwendet:

$$J(\mathbf{u}) = \frac{\mu_u}{2} \sum_{i=1}^{M} \mathbf{u}_i^2 + \frac{\mu_s}{2} (\mathbf{s}_{M+1} - \mathbf{s}_d)^2. \tag{11}$$

Sie dient der Optimierung eines Kontrollers, welcher das animierte Modell in einen vorgegebenen Endzustand $\mathbf{s}_d$ überführen soll. Der erste Term maximiert dabei die Effizienz des Kontrollers, während der zweite Term die Einhaltung der Positionsrestriktion $\mathbf{s}_d$ erzwingt.

Die Suche nach dem optimalen Kontroller erfolgt mittels einer Variante des Backpropagation-Algorithmus, dem *backpropagation through time*-Algorithmus:

$$\mathbf{u}^{l+1} = \mathbf{u}^l + \eta_x \nabla_{\mathbf{u}} J(\mathbf{u}^l). \tag{12}$$

Mit l wird hierbei die Iteration des Minimierungsschrittes bezeichnet und η_x steht für die verwendete Lernrate.

In einer ersten Phase berechnet der Algorithmus für jede Iterationsstufe die Zustandssequenz $\mathbf{s}^l = [\mathbf{s}_1^l, \mathbf{s}_2^l, \ldots, \mathbf{s}_{M+1}^l]$ gemäss (10) aus den Kontroll-Inputs $\mathbf{u}^l = [\mathbf{u}_1^l, \mathbf{u}_2^l, \ldots, \mathbf{u}_M^l]$. In einem nächsten Schritt berechnet er die Komponenten von $\nabla_{\mathbf{u}} J$ aus (12). Dies kann er auf eine effiziente Art und Weise, da die Struktur des Perzeptrons die Gradientenberechnung mittels der Kettenregel der Differentialrechung erlaubt. In der letzten Phase wendet er die errechneten Korrekturen im Gegensatz zum normalen Backpropagation-Algorithmus auf die *Eingabegrössen* und nicht auf die Verbindungsgewichte an — er justiert die Kontroller-Sequenz $\mathbf{u}$.

Modell-Beschreibung	Physik. Simulation	$\mathbf{N}_\Phi^{25}$	$\mathbf{N}_\Phi^{50}$	$\mathbf{N}_\Phi^{100}$
Passives Pendel	4.70	0.10	0.05	0.02
Aktives Pendel	4.52	0.12	0.06	0.03
Geländewagen	4.88	—	0.07	—
Lunar Lander	6.44	—	0.12	—
Delphin	63.00	—	0.95	—

Tabelle 1: Vergleich der Simulationszeiten zwischen physikalischer Simulation mit SD/FAST und verschiedener NeuroAnimatoren $\mathbf{N}_\Phi^n$. Jeder Test dauerte 20'000 Zeitschritte in der physikalischen Simulation.

Das eben umrissene Verfahren findet auf sehr effiziente Art und Weise Kontroller, welche den gestellten Anforderungen in grossem Masse nachkommen. Allerdings lässt sich das verwendete Gradientenabstiegs-Verfahren noch weiter beschleunigen, z.B. durch die Einführung eines sog. *Momentum-Terms*. Ein Vorteil der mit dem NeuroAnimator erzeugten Kontroller liegt darin, dass sie nicht nur für die Emulation der Modelle mit dem NeuroAnimator verwendet werden können, sondern auch für die Steuerung der originalen physikbasierten Simulationen.

6 Resultate

Nachdem bis jetzt v.a. die theoretische Seite des NeuroAnimators zu Sprache gekommen ist, sollen in diesem Abschnitt noch einige der von Grzeszczuk und Terzopoulos erzielten Ergebnisse betrachtet werden.

6.1 Ergebnisse der Emulation

Bei der Emulation der in Abb. 7 gezeigten physikbasierten Modelle hat sich gemäss [1] der NeuroAnimator der klassischen numerischen Simulation als deutlich überlegen gezeigt:

Performance

Die Emulation eines dynamischen Systems mit einem Zustandsvektor $\mathbf{s}$ der Grösse N benötigt nie mehr als $O(N)$ *hidden neurons*. Desshalb kann ein solcher Emulationsschritt mit $O(N^2)$ Operationen

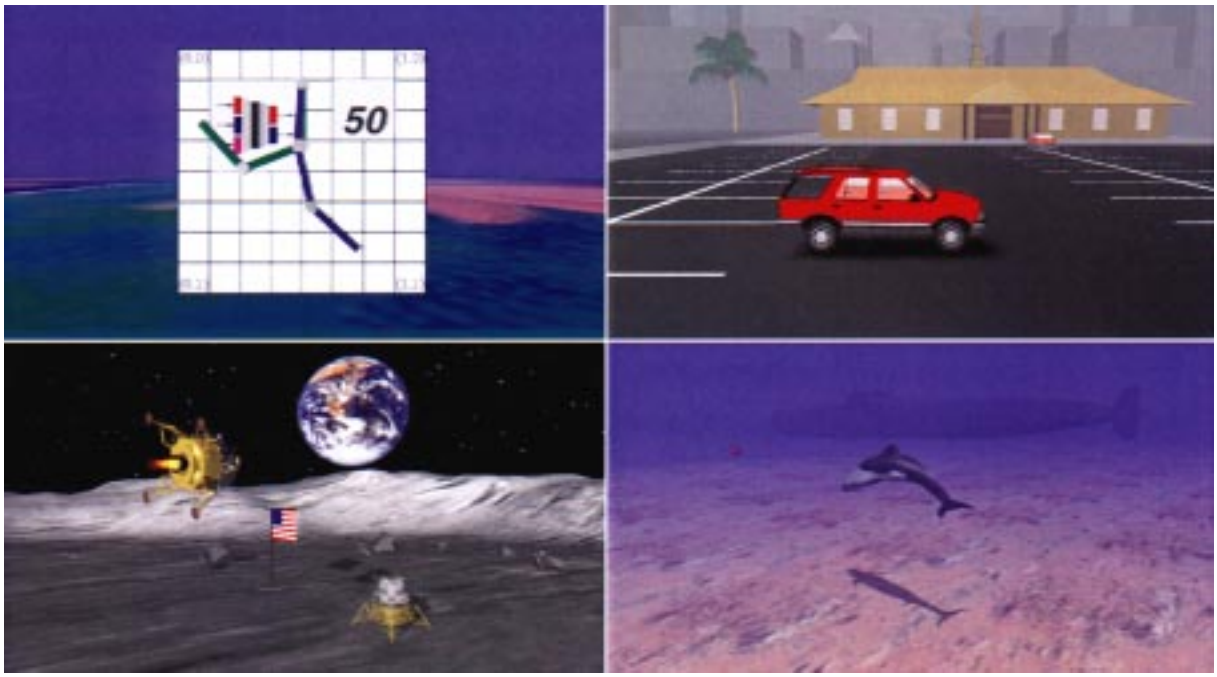

Abbildung 7: Anwendungen des NeuroAnimators: Das Bild oben links zeigt ein planares Multilink-Pendel und das Bild oben rechts den Emulator für einen parkenden Geländewagen. Unten links ist eine Momentaufnahme einer emulierten Mondlandung zu sehen und auf der Aufnahme unten rechts das Modell eines Delphins, wie es im Text beschrieben ist. (©1998 by R. Grzeszczuk)

evaluiert werden. Im Vergleich dazu kostet ein Simulationsschritt mit einem impliziten Integrationsverfahren $O(N^3)$ Operationen. Dabei darf nicht vergessen werden, dass ein einzelner Emulationsschritt oft gleichbedeutend mit bis zu 50 numerischen Simulationsschritten ist.

Tabelle 1 gibt einige Performance-Vergleiche zwischen numerischer Simulation und der Emulation mit neuronalen Netzwerken wieder. Die Notation $\mathbf{N}_\Phi^n$ steht dabei für einen NeuroAnimator der mit Super-Zeitschritten $\Delta t = n\delta t$ trainiert wurde. Am Beispiel des NeuroAnimators $\mathbf{N}_\Phi^{100}$ wird ersichtlich, dass die Emulation mit neuronalen Netzwerken gegenüber der physikalischen Simulation 50 bis 100 mal schneller ist.

Eine Möglichkeit zur weiteren Performance-Steigerung ebenso wie zur Verminderung der Netzwerkkomplexität, ist die hierarchische Strukturierung eines neuronalen Netzwerkes in mehrere Teilnetzwerke, die ihrerseits ein neuronales Netzwerk bilden. Beispielsweise könnte für jede der Gliedmassen eines Menschenmodells ein eigenes Teilnetzwerk eingesetzt werden. Ein weiteres Beispiel ist die hierarchische Gliederung des Fischmodells aus Abb. 2 gemäss den einzelnen Körpersegmenten.

Approximations-Fehler

Es hat sich gezeigt, dass der Fehler, der von einem NeuroAnimator erzeugten Approximation mit grösserwerdendem Super-Zeitschritt Δt nur unwesentlich zunimmt: Der Approximationsfehler eines Emulators mit Schrittweite Δt_s akkumuliert sich im Laufe von n Iterationen und bewegt sich schliesslich in der gleichen Grössenordnung wie der Approximationsfehler, welcher von einem Emulator mit Schrittweite $\Delta t_l = n\Delta t_s$ in einer Iteration erzeugt wird.

Der einzige Nachteil, der somit aus der Verwendung einer grossen Schrittweite Δt entstehen kann, ist ein allfälliger Verlust von Details in der Animation, z.B. hochfrequente Anteile der Bewegung. Allerdings trat dieses Phänomen bei den Experimenten von Grzeszczuk und Terzopoulos nicht auf, so dass nichts wirklich gegen die Verwendung grosser Iterationsschrittweiten spricht[12].

Ein weiterer Spezialfall bilden die deformierbaren Feder-Masse-Modell, welche bei langen Animationssequenzen (100'000 Emulationschritte und mehr)

[12]Selbstverständlich lässt sich die Schrittweite nicht beliebig erhöhen, da das neuronale Netzwerk irgendwann keine adäquate Approximation mehr finden kann.

einer Deformation unterworfen sind, welche durch den akkumulierten Approximationsfehler der Emulation entsteht. Diesem Problem kann aber durch eine periodische Korrektur des Modells begegnet werden.

Probleme neuronaler Netze

Selbstverständlich unterliegt der NeuroAnimator auch den üblichen Problemen neuronaler Netzwerke:

- Schlecht vorhersagbares Input/Output-Verhalten bei Eingaben die nie als Trainingsbeispiel gelernt wurden[13].
- Bei der Ausgabe handelt es sich generell um eine fehlerbehaftete Approximation.
- Das Netzwerk kann nicht oder nur sehr schwer analysiert werden.
- Gradienten-Abstiegs-Verfahren wie der Backpropagation-Algorithmus konvergieren langsam, können in lokalen Minimas "festhängen" oder oszillieren und sogar einmal gefundene gute Minimas wieder verlassen [4].

6.2 Ergebnisse der Kontroller-Synthese

Die Kontroller-Synthese mit neuronalen Netzwerken erwies sich in den Experimenten von Grzeszczuk und Terzopoulos ebenfalls als sehr effizient. In der Regel wurde der gewünschte Kontroller nach ca. 15 bis 20 Optimierungsschritten gefunden.

Exemplarisch soll die Effektivität der Neuro-Kontroller-Synthese am Beispiel des Fischmodells demonstriert werden: Die von Grzeszczuk und Terzopoulos ursprünglich verwendete, klassische Methode, um einen Kontroller für effektive Schwimmbewegungen eines Delphinmodells zu finden, benötigte 500 bis 3500 Lernschritte. Demgegenüber rechnet der Ansatz mit neuronalen Netzwerken gerade mal 20 Lernschritte um eine ebenbürtige Lösung zu finden. So richtig fassbar wird dieses Verhältnis, wenn man die benötigten Berechnungszeiten vergleicht: Der klassische Ansatz verschlingt über 1 Stunde Rechenzeit, um den Kontroller zu erzeugen — der NeuroAnimator weniger als 10 Sekunden.

[13] Es ist nicht garantiert, dass das neuronale Netzwerk "vernünftig" reagiert, insbesondere bei Eingaben, die ausserhalb des von den Trainingsbeispielen abgedeckten Bereichs liegen.

7 Schlussbemerkung

Natürlich ist es schwierig, sich eine eigene Meinung über ein Verfahren wie das des NeuroAnimators zu bilden, wenn man keine eigene Experimente und Performance-Messungen durchführen kann. Glaubt man aber den in [1] veröffentlichten Ergebnissen — und es gibt keinen Grund, dies nicht zu tun — so präsentiert sich der NeuroAnimator als äusserst effektiver Ansatz, um realistische Animationen physikbasierter Modelle zu erzeugen.

Ein grosser Vorteil des NeuroAnimators ist seine *off-line*-Charakteristik: Auch wenn der Aufwand für das Training des neuronalen Netzwerkes gegenüber dem reinen Simulieren der Animationen zunimmt, zahlt sich der betriebene Aufwand aus, wenn man das emulierte Modell wiederholt und mit unterschiedlichen Trajektorien animieren will. Die reine Emulation eines Modells mit einem trainierten NeuroAnimator ist um Grössenordnungen schneller, als die erneute Simulation des Modells mit klassischen Integrationsmethoden.

Auch auf dem Gebiet der *Artificial Lifeforms* kann der NeuroAnimator in Verbindung mit anderen, z.B. verhaltensbasierten Methoden, interessante Möglichkeiten eröffnen, um künstliche Lebensformen realistisch zu animieren.

Man darf, um zu einem Schluss zu kommen, wohl festhalten, dass die hier vorgestellte Methode zur Animation physikbasierter Modelle sehr attraktiv ist und weitere Forschungsbemühungen auf dem Gebiet lohnenswert sind.

Literatur

[1] R. Grzeszczuk, D. Terzopoulos, and G. Hinton. *NeuroAnimator: Fast Neural Network Emulation and Control of Physics-Based Models.* In *SIGGRAPH*, 1998.

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

[3] D. Terzopoulos, X. Tu, and R. Grzeszczuk. Artificial fishes: Autonomous locomotion, perception, behaviour, and learning in a simulated physical world. *Artificial Life*, 1(4):327–351, 1994.

[4] A. Zell. *Simulation Neuronaler Netze.* Addison-Wesley, 1994.