Machine Learning in Physically-Based Simulations

CS 686 Project

Instructor: Kate Larson

Author: Egor Larionov (20263767)

University of Waterloo

December 9, 2013

Abstract

This project introduces the applications of artificial intelligence and machine learning to the field of computer graphics, with an emphasis on physically based simulation. A non-exhaustive overview of various machine learning methods is provided with examples. An introduction to function approximation using artificial neural networks is provided. A 2D particle simulator is used to illustrate the applicability of learning techniques in physically-based simulations. A Matlab program is built to simulate 2D particles with an impulse based collision reaction function. An artificial neural network is then trained to approximate this function, and the results are then compared qualitatively and quantitatively.

1 Introduction

Artificial intelligence and machine learning has become a popular and important branch of computer science especially over the past few decades. As computers became faster and more accessible (think mobile devices and the internet), it is difficult to live a day in the modern age without interacting with an artificially trained system. Many common services are supported by machine learning, including automatic translation, voice recognition, object recognition and many more. Machine learning is especially useful for solving ill-posed problems or when the nature of a problem is not well known. For instance, simulating a physical phenomenon such as fire, smoke, water or even a falling leaf, is a problem especially difficult to simulate due to the complexity of its computational model.

Simulation of natural phenomena can have many purposes. It may be essential to replicate a physical phenomenon for scientific or engineering purposes such as creating an accurate model of air flow for weather forecasting. On the other hand, physical phenomena play an important role in computer graphics, where creating an aesthetically pleasing simulation is a priority over accuracy. In this area, machine learning is very applicable, because we only want to approximate a physical phenomena to a point where it is indistinguishable from the "real thing" to a human observer. In some cases the approximation can be coarse (fire and smoke), while in others it must be more accurate (walking, facial expressions), depending on human familiarity with the phenomena. We see the way other people walk and talk every day, which enables us to detect the slightest flaw in a walking or speech simulation. However explosions, fire and smoke are not common in our everyday life, so we may be more easily fooled by an inaccurate graphical representation of these phenomena.

This paper will introduce a few contemporary learning problems, and indicate a few of their uses in physically-based animation research. I will focus on a particular method for function learning called *artificial neural networks*. To illustrate the applicability of neural networks, I will approximate an impulse reaction function used in a simple 2D particle simulator, with the help of Matlab's built-in neural network simulator.

2 Machine Learning

Machine learning, a branch of artificial intelligence, is the study of methods that enable computers to learn, and perform tasks without being explicitly programmed to do so. Machine learning methods are widely applied to problems computer graphics, many of which include: regression, function approximation, manifold learning, learning based optimization and data-driven classification [1, 2.2]. We will focus on the first three of these methods, since they are more applicable to physical simulation problems.

2.1 Regression

Suppose there is an unknown stochastic process that can be observed by some variable known to be dependent on some k independent variables[1]. So the process can be described [3] by a function f such that

$$y = f(\mathbf{x}, \mathbf{p}) + e$$

where y is the response (observed) variable, $\mathbf{x} \in \mathbb{R}^k$ are the independent variables, e are the errors (usually assumed to be normally distributed), and $\mathbf{p} \in \mathbb{R}^m$ are some unknown parameters. We are often interested in finding the relationship between the input and the output. So we may take some M independent measurements (samples) of this process, $A = \{\mathbf{x}_i\}_{i=1}^M$, and $\mathbf{y} = \{y_i\}_{i=1}^M$, then use a

[1]There exist generalizations of regression to multidimensional dependent variable.

function that will mimic the behaviour of f:

$$\tilde{f}:\ \underbrace{\mathbb{R}^{M\times k}}_{\substack{\textit{sample}\\ \textit{space}}} \times \underbrace{\mathbb{R}^m}_{\substack{\textit{parameter}\\ \textit{space}}} \to \mathbb{R}^M,$$

and find parameters $\mathbf{p}$ that will approximate f. Then $\tilde{f}$ may reproduce the process up to an appropriate error margin as:

$$\mathbf{y} = \tilde{f}(A, \mathbf{p}) + \mathbf{e}.$$

For instance we may throw a ball in the air and measure two changing variables: its height h and the time of measurement t. Using a linear regression technique for simplicity[2], we may assume that the height of the ball depends linearly on t and t^2. With a set of measurements $A = [\mathbf{x}_1|\dots|\mathbf{x}_M]^T$, where $\mathbf{x}_i = (t_i, t_i^2)^T$ and $\mathbf{y} = (h_1, \dots, h_M)^T$ (here ^{T} denotes transposition), it remains to find a set of parameters $\mathbf{p} \in \mathbb{R}^2$ such that

$$h_i = p_1 t_i + p_2 t_i^2 + e_i \qquad \text{for all } i \in \{1, \dots . M\}$$

where e_i are minimized. In vector form we get

$$\mathbf{y} = \begin{bmatrix} \mathbf{x}_1 \\ \hline \vdots \\ \hline \mathbf{x}_M \end{bmatrix} \begin{bmatrix} p_1 \\ p_2 \end{bmatrix} + \begin{bmatrix} e_1 \\ \vdots \\ e_M \end{bmatrix} = A\mathbf{p} + \mathbf{e} = \tilde{f}(A, \mathbf{p}) + \mathbf{e}$$

where $\tilde{f}(A, \mathbf{p}) = A\mathbf{p}$ is given by the linear regression assumption.

2.2 Function Approximation

This problem is closely related to regression. As with regression problems, we try to find an approximation to some unknown function, whose behaviour we can measure. The main difference is that function approximation methods focus on trying to approximate arbitrarily complex functions numerically, while regression attempts to fit a known simpler function to a set of data by finding the right parameters. Usually regression techniques are applied to functions of stochastic nature, while function approximation is used to match deterministic functions.

This time we will define the problem for functions with multidimensional outputs. Suppose we would like to approximate some unknown function

$$f : \mathbb{R}^m \to \mathbb{R}^n,$$

using a set of measurements of its inputs and outputs $\{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^M$ where $\mathbf{x}_i \in \mathbb{R}^m$ and $\mathbf{y}_i \in \mathbb{R}^n$. This can be done by interpolating this set of measurements in a clever way. The differences in function approximation methods include (dimensional) scalability, difficulty of implementation and rate of convergence (how many samples are need to approximate a function within a small enough error).

There are many computational used techniques to solve regression and function approximation problems. A few popular techniques are listed in [1]:

k-nearest neighbour: A form of instance-based learning [6, 20.4], that directly interpolates input-output examples by estimating the probability density of each neighbourhood containing k neighbours. This method is scalable to multiple dimensions and easy to implement, however it requires lots of storage.

[2]Assuming our model is linear is not necessary, since there exist more general polynomial (e.g. [4]) or statistical regression models (see [5] for details)

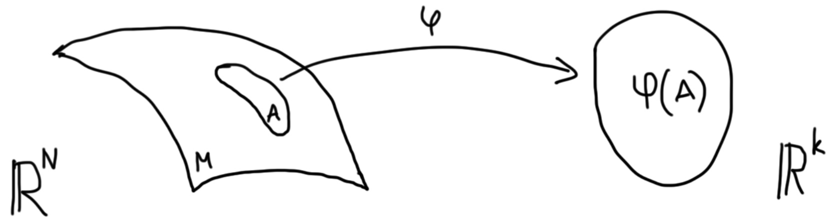
artificial neural networks: A computational model of a network of nodes resembling the neural network in a brain. Each node i in the network is a multidimensional function, which outputs a value between 0 and W_i (the weight at the given node) depending on how the "strength" the weighted average of its inputs coming from other nodes. The weights of the network are adjusted at each sample input, such that eventually the network approximates the target function. This method works well for noisy inputs and is easy to implement, however it lacks in scalability and can only approximate smooth functions. Artificial neural networks will be explored in detail in a later section.

radial basis functions (RBF): A popular kind of probabilistic artificial neural network, which trains (the means and variances of) Gaussian probability distributions that define the weights at each node in the network.

2.3 Manifold Learning

Manifold learning can be seen as a generalization to function approximation. The assumption that $f : \mathbb{R}^m \to \mathbb{R}^n$ is a well-defined function is lifted, and instead input-output examples used to learn f are seen as points in $\mathbb{R}^N$ (where $N = n \times m$) that define a k-manifold where $k < N$.

Loosely speaking, a *k-manifold M*, is a subspace of $\mathbb{R}^N$ such that for every point on $p \in M$ there exists a neighbourhood $A \subset M$ homeomorphic[3] to an open subset $V \subset \mathbb{R}^k$.



Intuitively, a 2-manifold in a 3D space is a smooth surface which looks planar when inspected closely at a point. For instance the surface of the earth is (roughly) a sphere, however to us it appears flat because we can only see a small portion of it at once.

One example of manifold learning in computer graphics is approximating a 3D surface with a smooth closed surface. Often surfaces used in modelling 3D objects for visual purposes contain holes, which can be automatically repaired with manifold learning algorithms [8].

Manifold learning is more generally referred to as nonlinear dimensionality reduction, and there are many techniques available to solve this problem. A few include [1]:

principle component analysis (PCA): A popular statistical method that reduces dimensionality of input data points linearly by projecting them onto a lower dimensional space. See [9] for a thorough analysis of PCA. This method is numerically stable, however it does not work for non-linear problems.

kernel PCA: Non-linear extension of PCA. This method is accurate, however naturally slower than standard PCA.

self-organizing map (SOM): A kind of artificial neural network, that fits a discrete grid to a set of unorganized data points. This method is fast and easy to implement, however it is not widely applicable.

[3] Two open sets $A \in \mathbb{R}^N$ and $B \in \mathbb{R}^k$ are homeomorphic if there exists a continuous bijection $\varphi : A \to B$ with a continuous inverse, φ^{-1}. See your favourite book on mathematical analysis [7] for details.

3 Modelling Physical Systems for Animation

Animation of physical systems often suffers from the sheer complexity of some physical models. In addition, some systems are so visually complex that it is exceptionally difficult (or impossible) to accurately model them by hand, as artists often create character animation. Thus it is truly a challenge to design efficient and stable algorithms to perform physical simulation, while exposing intuitively sensible parameters to the user.

Many of these problems can be relieved by machine learning. To create aesthetically pleasing animation, it is sufficient to merely approximate a physical system, thus avoiding the need for computing an explicit physical model. It is usually more efficient to approximate a function than compute it exactly. Furthermore machine learning methods can expose custom user-modifiable parameters that are not necessarily linked to physical properties, which is sometimes more useful for artists. Finally, machine learning solutions are often easier to implement than explicit physical simulators.

There are a variety of physically-based simulation problems involving deformable objects (e.g. cloth, hair, pillows), stochastic systems (e.g. candle flame, falling leaf) and incompressible fluids (e.g. explosions, water) as outlined in [1, 4.4]. All of these problems are computationally difficult to solve, but can be effectively approximated with machine learning techniques. One particularly popular example of a general method for solving physically-based problems, is an artificial neural network method, used for function approximation, called the*NeuroAnimator* [2]. The NeuroAnimator is designed to produce subsequent states of a physical system given a number of control inputs at every time step. The output from each time step is fed into the network for the subsequent time step, along with time dependent controls, and forces. This can be written as $\mathbf{s}_{t+\Delta t} = \mathcal{N}(\mathbf{s}_t, \mathbf{u}_t, \mathbf{f}_t)$, where $\mathcal{N}$ is the neural network, $\mathbf{s}_t$ is the current state, $\mathbf{u}_t$ are the controls, and $\mathbf{f}_t$ are the forces acting on the system at time t. The main advantages of this method is its performance and generality, however it suffers from lack of scalability to multiple dimensions. Furthermore, neural networks can only approximate smooth functions, so care must be taken when choosing functions to be approximated. Note that the state alone has as many dimensions as there are degrees of freedom of the object being simulated. This method uses the back-propagation learning algorithm, similar to the one used in Matlab, which I will leverage in simulating a 2D particle collision system.

4 A 2D Particle Collision Simulator

A simple problem I explored was the simulation of moving particles in a box in two dimensions. All particles are assumed to have equal mass and resemble balls (circles in 2D) of radius r. The containing box has dimensions $N \times N$. There are M particles inside the box, initially at some random positions $\{\mathbf{p}_i = (x_i, y_i)\}_{i=1}^{M}$ and random velocities $\{\mathbf{v}_i = (v_x, v_y)_i\}_{i=1}^{M}$. At each consecutive simulation frame, the particles are redrawn to be at new positions $\mathbf{p}'_i = \mathbf{p}_i + \mathbf{v}_i$ with unchanged velocities, unless they have collided with a wall or a neighbouring particle. All collisions are assumed to be perfectly elastic. A particle at position (x, y) is collided with a wall if

$$\begin{array}{llll} x > N - r & \text{or} & x > r & \text{for right and left walls, or} \\ y > N - r & \text{or} & y > r & \text{for top and bottom walls resp.} \end{array}$$

In a case of a wall collision, the velocity of the particle simply changes sign at the appropriate coordinate. For instance if a particle with velocity (v_x, v_y) collides with the left or right wall, its velocity changes to $(-v_x, v_y)$.

4.1 Physical 2D Collision Model

Collisions between particles are slightly more complicated. However they can be easily expressed as a function that takes the state of particles before collision to the state after collision. From a physical perspective, a state of two particles can be fully described by a point in the 2D phase space before and after a collision.

Given some stationary frame of reference, two particles with centres at positions $\mathbf{p}_1$ and $\mathbf{p}_2$ are collided when $\|\mathbf{p}_1 - \mathbf{p}_2\|_2 < 2r$, (where $\|\cdot\|_2$ is the standard euclidean norm). The initial velocities of the two particles are denoted by $\mathbf{v}_1$ and $\mathbf{v}_2$. After the collision, the particles will have velocities $\mathbf{u}_1$, and $\mathbf{u}_2$. The collision reaction can be expressed as a function $f : \mathbb{R}^8 \to \mathbb{R}^4$ such that $(\mathbf{u}_1, \mathbf{u}_2) = f(\mathbf{p}_1, \mathbf{p}_2, \mathbf{v}_1, \mathbf{v}_2)$. Finally, to insure that the particles do not remain in the state of collision at the next frame, each particle should be displaced away from the other by adding (or subtracting) a vector $\mathbf{b}$ to (or from) each particle, so the collision is really given by $f : \mathbb{R}^8 \to \mathbb{R}^6$, where

$$(\mathbf{u}_1, \mathbf{u}_2, \mathbf{b}) = f(\mathbf{p}_1, \mathbf{p}_2, \mathbf{v}_1, \mathbf{v}_2).$$

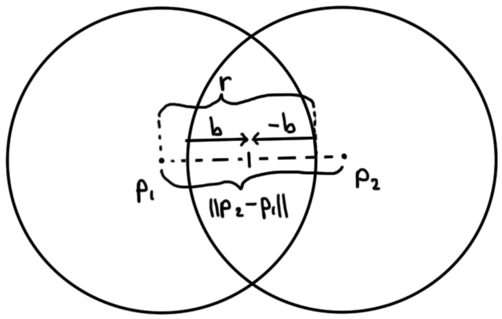
The goal of this example is to learn the function f. Note that f is a deterministic, well-defined function, so a function approximation method is sufficient to learn f. I chose to generate an artificial neural network to simulate this function.

In order to train the neural network, we need a set of samples. Ideally we would not need to implement the physical model to train my neural network, and instead provide a segmented recording of real colliding spheres (e.g. a game of billiards). However for the sake of simplicity I provided an explicit collision function to train the neural network. This function will be used to determine the error in the output of our neural network. The collision function is given by

$$\mathbf{u}_1 = \mathbf{v}_1 + (\mathbf{v}_r \cdot \mathbf{n})\,\mathbf{n} \tag{4.1}$$

$$\mathbf{u}_2 = \mathbf{v}_2 - (\mathbf{v}_r \cdot \mathbf{n})\,\mathbf{n} \tag{4.2}$$

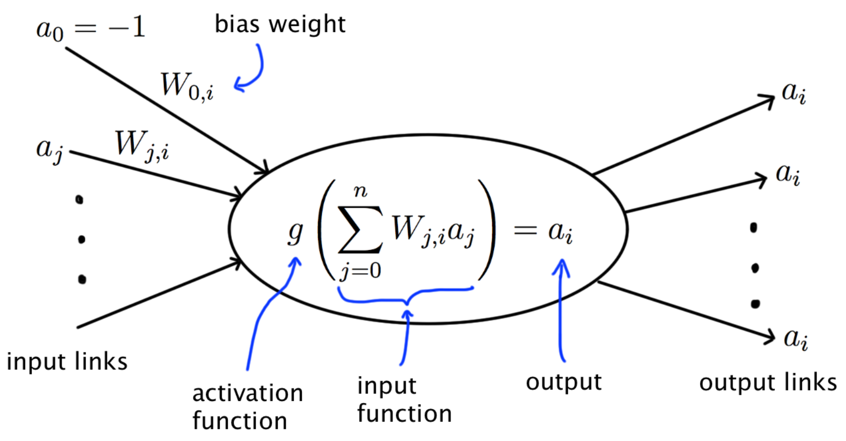
$$\mathbf{b} = r - \tfrac{1}{2}\|\mathbf{p}_1 - \mathbf{p}_2\| \tag{4.3}$$



where r is the radius of each circular particle, $\mathbf{v}_r = \mathbf{v}_2 - \mathbf{v}_1$ is the relative velocity of the two particles, and $\mathbf{n} = (\mathbf{p}_2 - \mathbf{p}_1)/\|\mathbf{p}_2 - \mathbf{p}_1\|$. Note that after collision, the particles are moved to positions $\mathbf{p}'_1 = \mathbf{p}_1 - \mathbf{b}$, and $\mathbf{p}'_2 = \mathbf{p}_2 + \mathbf{b}$.

4.2 Feed-Forward Artificial Neural Network

Recall that an artificial neural network is composed of nodes (or units) connected by directed links. Each node i computes an activation value, a_i, and propagates it to the rest of the network through output links, each having a weight $W_{j,i}$, where j is the index of the destination node. A node has the following form:



where the activation function g takes a value near zero if the signal (weighted linear combination of inputs given by the input function) is weak (negative), and 1 if the signal is strong (positive). This means usually takes the form $g(x) = 1/(1+e^{-\alpha x})$, where $\alpha \in (0, \infty]$, and at $\alpha = \infty$, g becomes the threshold function[4]. The bias weight determines the location of the threshold (at which point g is $\frac{1}{2}$ or changes value in case $\alpha = \infty$). A *feed-forward* network contains no back links, meaning that one node can not affect any of its ancestors. This type of network suits the problem at hand very well. The initial input units a_i to a neural network are the inputs to the function being approximated. Similarly the output units represent the outputs of the function. Between the input and output layers in the network, there may be an arbitrary number of hidden layers, with an arbitrary number of units in each layer.

One major challenge in designing effective neural networks is determining the number of hidden layers and hidden units to use. It has been shown [10] that for full generality, it is sufficient to have two hidden layers in a feed-forward network. It has also been suggested that having two hidden layers can be advantageous [11]. I train the neural network with various configurations, and analyze the error in position and velocity.

4.3 Results

I trained and simulated the neural network on a particle simulator with an 256×256 pixel grid with 5 particles of 20 pixel radii. The initial velocity of the particles was capped at 8 pixels per frame. Each simulation contained 5000 frames.

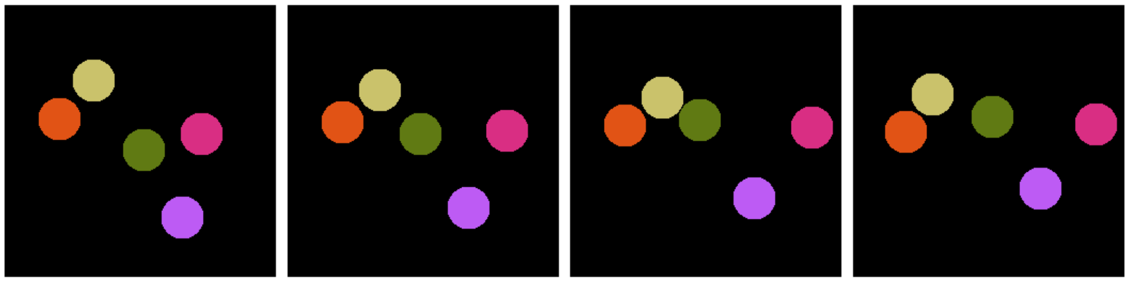


Figure 4.1: A sample of 4 frames from the proposed simulation.

All simulations were run on the University of Waterloo undergraduate computing environment

[4]Note that if g is linear than the neural network becomes the linear regression technique.

using all available CPUs. I trained 18 different neural networks, with varying hidden layer structure and training sets. The following errors were found in the velocity function in each neural network:

# of units		size of training set		
Layer 1	Layer 2	76	324	672
5	5	2.377 (289)	1.357 (300)	1.273 (288)
10	5	1.199 (131)	0.3152 (314)	0.6532 (303)
10	10	4.558 (652)	0.2632 (275)	0.7738 (288)
20	15	8.271 (1509)	0.1559 (336)	0.08141 (412)
5	0	2.236 (317)	1.543 (278)	1.342 (271)
10	0	1.820 (383)	0.9019 (309)	0.407 (303)
20	0	2.140 (501)	0.1298 (341)	0.09193 (380)
25	0	-	0.1120 (336)	0.08796 (376)
35	0	-	0.1483 (347)	0.09733 (360)

Table 4.1: Average error in response velocity after collision over all encountered collisions, as computed by $e_v = \frac{1}{k}\sum_{t=1}^{k}\frac{1}{2}(\|\mathbf{u}_1 - \mathbf{w}_1\| + \|\mathbf{u}_2 - \mathbf{w}_2\|)$, where $\mathbf{w}_i$ are the post-collision velocities as computed by the explicit model in (4.1) and (4.2), and k is the total number of collisions occurred (displayed in parentheses beside each error).

# of units		size of training set		
Layer 1	Layer 2	76	324	672
5	5	1.394 (289)	0.8881 (300)	0.7904 (288)
10	5	0.6596 (131)	0.9759 (314)	0.9372 (303)
10	10	2.6051 (652)	0.6147 (275)	0.6621 (288)
20	15	6.0223 (1509)	0.3126 (336)	0.3356 (412)
5	0	0.9082 (317)	0.7961 (278)	0.7882 (271)
10	0	1.115 (383)	0.7176 (309)	0.5354 (303)
20	0	3.310 (501)	0.3202 (341)	0.2936 (380)
25	0	-	0.3136 (336)	0.2200 (376)
35	0	-	0.3475 (347)	0.2708 (360)

Table 4.2: Average error in response position after collision over all encountered collisions, as computed by $e_p = \frac{1}{k}\sum_{t=1}^{k}\frac{1}{2}(\|\mathbf{p}'_1 - \mathbf{q}'_1\| + \|\mathbf{p}'_2 - \mathbf{q}'_2\|)$, where $\mathbf{q}'_i$ are the post-collision positions as computed by the explicit model in (4.3), $\mathbf{p}'_i$ are the post-collision positions computed from the neural network, and k is the total number of collisions occurred (shown beside each error).

From the tables above, it is evident that the error in position stabilizes for each given neural network given a big enough training set (larger than 324). Furthermore the error decreases with an increase in the number of units. Furthermore notice that increasing the number of units (over 20) in the first layer fails to improve accuracy, however if the units are distributed along two layers, the accuracy is improved when the total number of units increases.

The algorithm used by MATLAB to train the neural networks is Levenberg-Marquardt backpropagation, which stops when the network seizes to change.

I have include the times it took for each neural network to attain its best accuracy for completeness. However, tells us little about the relationship between number of units and training time because each network was trained only once. To obtain a better estimate on this relationship multiple trials must be performed, the average of which may provide some information.

# of units		size of training set		
Layer 1	Layer 2	76	324	672
5	5	16.94	63.45	70.29
10	5	9.544	176.54	68.78
10	10	8.850	61.33	16.06
20	15	23.84	117.2	697.6
5	0	1.824	6.206	10.84
10	0	6.430	46.09	85.95
20	0	3.381	101.4	129.8
25	0	-	53.66	84.84
35	0	-	41.76	120.8

Table 4.3: CPU time spent training each neural network in seconds.

4.4 Generated Output

Each of the artificial neural networks in the results produced a video of the simulation using `ffmpeg` to combine frames into an `.mpeg` file. The number of frames simulated varies.

`imvnetsmall[num_neurons]_[frames_trained].mpg`: All video files of simulated artificial neural networks used, where `num_neurons` is the number of units used in each layer (e.g. `5-5`) and `frames_trained` is the number of frames of the physical simulation used to train each neural network (`1000`, `5000` or `10000`).

`imvnetcolor[num_neurons]_[frames_trained].mpg`: Video files of neural network simulations performed on a 512×512 grid, with 20 particles. Parameters are as above.

`imphy.mpg`: A 2D particle simulation on a 512×512 grid and 20 particles using the physical collision function in (4.1), (4.2), and (4.3).

`imphysmall.mpg`: A 2D particle simulation on a 256×256 grid and 5 particles using the physical collision function in (4.1), (4.2), and (4.3). This video was create with the following commands

```
>> sp(256,20,5,8,2000,3,@bbcollision);
>> system('ffmpeg -r 40 -f image2 -i "imv-%05d.png" -b 64k -y "imphy.mpg"');
```

`ploterrpsm[num_neurons]_[frames_trained].pdf`: Plots giving the errors as computed in Tables 4.2 and 4.1 of respective errors at each collision. The parameters are as above.

5 Conclusions

Research in machine learning techniques applied to computer graphics and specifically physically-based animation is extensive, however each method has its strengths and weaknesses. Artificial neural network are exceptionally powerful and widely applicable for low dimensional problems. Designing the most effective networks, however, has proven to be a challenge. In particular, it is not always clear how many units to use in each layer of a feed-forward neural network. Furthermore, the target system being approximated must be smooth, as otherwise neural networks will not produce the correct results. This is precisely the reason why I chose to use the collision function in the 2D particle simulator, because the velocity and position functions are discontinuous as functions of time. They are, however, smooth as functions of initial positions and velocities at a collision.

References

[1] J. Dinerstein, P. K. Egbert, and D. Cline. *Enhancing computer graphics through machine learning: a survey.* The Visual Computer, 23(1), pp. 25–43 (2007). ISSN 0178-2789. doi: 10.1007/s00371-006-0085-4.

[2] R. Grzeszczuk, D. Terzopoulos, and G. Hinton. *Neuroanimator: fast neural network emulation and control of physics-based models.* In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '98, pp. 9–20. ACM, New York, NY, USA (1998). ISBN 0-89791-999-8. doi:10.1145/280814.280816.

[3] Gallant. *Nonlinear regression.* The American Statistician, 29(2), pp. 73–81 (1975). doi:10.1080/00031305.1975.10477374.

[4] H. Theil. *A rank-invariant method of linear and polynomial regression analysis.* In B. Raj and J. Koerts, eds., *Henri Theils Contributions to Economics and Econometrics*, vol. 23 of *Advanced Studies in Theoretical and Applied Econometrics*, pp. 345–381. Springer Netherlands (1992). ISBN 978-94-010-5124-8. doi:10.1007/978-94-011-2546-8_20.

[5] K. V. Mardia, J. Kent, and J. Bibby. *Multivariate analysis (probability and mathematical statistics)* (1980).

[6] S. J. Russell, P. Norvig, J. F. Canny, J. M. Malik, and D. D. Edwards. *Artificial intelligence: a modern approach*, vol. 74. Prentice hall Englewood Cliffs (1995).

[7] V. A. Zorich. *Mathematical analysis II*, vol. 2. Springer (2004).

[8] T.-Q. Guo, J.-J. Li, J.-G. Weng, and Y. ting Zhuang. *Filling holes in complex surfaces using oriented voxel diffusion.* In *Machine Learning and Cybernetics, 2006 International Conference on*, pp. 4370–4375 (2006). doi:10.1109/ICMLC.2006.259087.

[9] I. Jolliffe. *Principal component analysis.* Wiley Online Library (2005).

[10] E. D. Sontag. *Feedforward nets for interpolation and classification.* Journal of Computer and System Sciences, 45(1), pp. 20–48 (1992).

[11] D. L. Chester. *Why two hidden layers are better than one.* In *Proceedings of the international joint conference on neural networks*, vol. 1, pp. 265–268 (1990).

A Matlab code

The following code is written entirely by me. The following Matlab functions create and train the 2D particle collision simulator on a number of different neural networks.

A.1 Primary functionality

Explicit physical ball-to-ball collision function implementing equations (4.1), (4.2) and (4.3).

```
function [pos vel bitn] = bbcollision(p1, p2, v1, v2, r)
% compute a collision reaction between two particles at positions p1
% and p2, with velocities v1 and v2 respectively
vel = zeros(2,1);
pos = zeros(2,1);

n = p2 - p1;
d = norm(n);
n = n/d;
vr = v2 - v1;
vr_dot_n = dot(vr, n)*n;
vel(:,1) = v1 + vr_dot_n;
vel(:,2) = v2 - vr_dot_n;

% separate the balls so they don't get stuck together
bit = r - 0.5*d;
bitn = floor(bit)*n;
pos(:,2) = round(p2 + bitn);
pos(:,1) = round(p1 - bitn);
```

Explicit physical ball-to-wall collision function.

```
function [pn v] = wbcollisions(pn, v, N, r)
% determine if a particle in pn collides with the wall, if so, adjust its
% velocity in v, and adjust its displacement as to not go through the wall

len = length(pn);
for i = 1:len % check for wall collisions
  if (pn(1,i) > N - r) || (pn(1,i) < r) % right or left wall collision
    v(1,i) = -v(1,i); % mirror velocity vector

    % adjust positions to avoid particles visually passing the boundary
    if pn(1,i) > N - r
      pn(1,i) = N - r;
    end
    if pn(1,i) < r
      pn(1,i) = r;
    end
  end % if x coordinate not ok

  if (pn(2,i) > N - r) || (pn(2,i) < r) % bottom or top wall collision
    v(2,i) = -v(2,i); % mirror velocity vector

    % adjust positions to avoid particles visually passing the boundary
    if pn(2,i) > N - r
      pn(2,i) = N - r;
    end
    if pn(2,i) < r
      pn(2,i) = r;
    end
  end % if y coordinate not ok
 end
```

Function to perform the main simulation based on the given collision function.

```
function [errp, errv] = sp(N,r,M,maxv,frames,seed,bbcol)
% 2D particle simulator
% N -> resolution
% r -> radius of each particle in pixels
% M -> number of particles
% maxv -> maximum velocity (displacement per frame) in each direction
% frames -> number of frames to simulate
% seed -> a random seed used to produce random colors and initial conditions
% bbcol -> the collision response function used to resolve collisions
rand('seed', seed);

% generate a few particles
p = genparticles(N,r,M,maxv);

% generate a random color for each particle
c = gencolors(M);
f = fillcolorpoints(p, r, N);
I = fillcolorimage(f, N, c);

% assign random initial velocity vectors:
v = 2.0*maxv*(rand(size(p)) - 0.5);

imwrite(I, 'img/imv-00000.png'); % write initial image

% animate
pn = p;
len = length(pn);
collisions = 0;

% gather error information
errp = zeros(1, frames);
errv = zeros(1, frames);

tic;
for t = 1:frames
  pn = round(pn + v);

  % check if two particles intersect, if so change velocity
  for i = 1:(len-1)
    for j = (i+1):len
      if norm(pn(:,i) - pn(:,j)) < r+r-2 % ball-ball collision detected
        collisions = collisions + 1;

        [pos vel] = bbcol(pn(:,i), pn(:,j), v(:,i), v(:,j), r);

        % compute the error as compared to the actual physical simulator
        [pos2 vel2] = bbcollision(pn(:,i), pn(:,j), v(:,i), v(:,j), r);
        errp(collisions) = 0.5*(norm(pos(:,2) - pos2(:,2)) + norm(pos(:,1) - pos2(:,1)));
        errv(collisions) = 0.5*(norm(vel(:,2) - vel2(:,2)) + norm(vel(:,1) - vel2(:,1)));

        pn(:,i) = pos(:,1);
        pn(:,j) = pos(:,2);
        v(:,i) = vel(:,1);
        v(:,j) = vel(:,2);

      end % if ball-ball collision
    end % for each particle j
  end % for each particle i

  [pn v] = wbcollisions(pn, v, N, r);

  fn = fillcolorpoints(pn, r, N);
  In = fillcolorimage(fn, N, c);
  imwrite(In, sprintf('img/imv-%05d.png', t));
end
toc % display elapsed simulation time

errp(collisions+1:end) = [];
errv(collisions+1:end) = [];

collisions % display number of ball-ball collisions
```

Program used to simulate the explicit physical particle simulator to prepare the training set for neural networks.

```
function [input, target] = genvinput(N,r,M,maxv,frames)
% 2D particle simulator that generates an input and target vectors to be used
% in a collision reaction learning algorithm
% N -> resolution
% r -> radius of each particle in pixels
% M -> number of particles
% maxv -> maximum velocity (displacement per frame) in each direction
% frames -> number of frames to simulate
rand('seed', 1);

p = genparticles(N,r,M,maxv);

% assign random velocity vectors:
v = 2.0*maxv*(rand(size(p)) - 0.5);

% prepare for frame loop
pn = p;
len = length(pn);
collisions = 0;

% allocate space for output vectors
input = zeros(6, frames);
target = zeros(6, frames);

for t = 1:frames
  pn = round(pn + v); % update positions according to current velocity

  % check if two particles intersect, if so change velocity
  for i = 1:(len-1)
    for j = (i+1):len
      if norm(pn(:,i) - pn(:,j)) < r+r-2 % ball-ball collision detected
        collisions = collisions + 1; % count number of collisions

        rel = pn(:,j) - pn(:,i); % relative posn of j (from i)
        input(:,collisions) = [rel; v(:,i); v(:,j)]; % phase before the collision

        % compute ball-ball collisions
        [pos vel bitn] = bbcollision(pn(:,i), pn(:,j), v(:,i), v(:,j),r);
        pn(:,i) = pos(:,1);
        pn(:,j) = pos(:,2);
        v(:,i) = vel(:,1);
        v(:,j) = vel(:,2);

        target(:,collisions) = [bitn; v(:,i); v(:,j)]; % phase after collision
      end % if ball-ball collision detected
    end % for each ball
  end % for each ball

  [pn v] = wbcollisions(pn, v, N, r); % compute all wall-ball collisions
end

% clean trailing zeros
input(:,collisions+1:end) = [];
target(:,collisions+1:end) = [];

collisions % display the number of collisions
```

A script used to train a feed-forward network using the Levenberg-Marquardt backpropagation method.

```
function net = prepnet(numneurons, x, y)
% prepare network for simulation
% ( essentially train network with numneurons hidden nodes )
net = feedforwardnet(numneurons');
net = configure(net, x, y);
net = train(net, x, y);
```

A wrapper for `sp.m` used to simulate a trained network.

```
function [errp, errv] = simvnet(N,r,M,maxv,frames,seed,net)
% 2D particle simulator using a learned collision function
% N -> resolution
% r -> radius of each particle in pixels
% M -> number of particles
% maxv -> maximum velocity (displacement per frame) in each direction
% seed -> a random seed used to produce random colors and initial conditions
% net -> learned collision function

bbcol = @(p1, p2, v1, v2, r) bbcolnet(p1,p2,v1,v2,net);
[errp errv] = sp(N,r,M,maxv,frames,seed,bbcol);

end % simvnet

function [pos vel] = bbcolnet(p1, p2, v1, v2, net)
  % use the learned function for collision response
  res = net([p2 - p1; v1; v2]);
  bitn = res(1:2);
  vel = reshape(res(3:6), 2, 2);
  pos(:,2) = round(p2 + bitn);
  pos(:,1) = round(p1 - bitn);
end % bbcolnet
```

Program that generates the training set, trains a network and simulates the 2D particle collision simulator.

```
function [net, errp, errv] = svn(N,r,M,maxv,ftrain,fsim,numneurons)
% Program that trains a neural network the impulse reaction function in
% a 2D particle collision simulator, and then simulates fsim frames
% of colliding particles using this neural network generatively.
% in addition this program will create a plot of the error in velocity and
% position after the collision, and display error means
% INPUTS:
% N -> resolution
% r -> radius of each particle in pixels
% M -> number of particles
% maxv -> maximum velocity (displacement per frame) in each direction
% ftrain -> number of frames to train the neural network with
% fsim -> number of frames to simulate on output
% numneurons -> number of neurons in each layer to use on the network
% OUTPUTS:
% net -> the trained nural network
% errp -> the error in position at each collision
% errv -> the error in velocity at each collision

disp('Generating input...');
[x y] = genvinput(N,r,M,maxv,ftrain);
disp('Training neural network...');
net = prepnet(numneurons, x, y);
disp('Simulating...');
[errp, errv] = simvnet(N,r,M,maxv,fsim,3,net);
ploterr(errp, errv, numneurons, ftrain);
fprintf('velocity error mean = %f\n', mean(errv));
fprintf('position error mean = %f\n', mean(errp));
```

A.2 Supporting functionality

The script run to generate Tables 4.1, 4.2 and 4.3. This script shows how to run the above primary functions. In order to run this script, the caller must have created a subdirectory `/img/` and have access to the program `ffmpeg` in order to generate `.mpeg` videos of the simulations.

```
% Driver script used to generate al
N = 256; % resolution
r = 20; % particle radius
M = 5; % number of particles
maxv = 8; % maximum velocity of generated particles
fsim = 5000; % number of frames to use in simulation
for numneurons = [5 5 10 20 5 10 20 25 35; 5 10 10 15 0 0 0 0 0]
  numneurons
  for ftrain = [1000 5000 10000] % generates 76, 324 and 672 collisions resp.
    ftrain
    [net, errp, errv] = svn(N,r,M,maxv,ftrain,fsim,numneurons);
    disp('Converting frames to an mpeg...');
    nn = [num2str(numneurons(1)) '-' num2str(numneurons(2))];
    system(['ffmpeg -r 40 -f image2 -i "img/imv-%05d.png" -b 64k -y "img/imvnetsmall' ...
       nn '_' num2str(ftrain) '.mpg"']);
  end
end
```

A supporting program that finds the points within r of the centres of given particles.

```
function f = fillcolorpoints(p, r, N)
% given a set of coordinates in p, generate the coordinates in the image that
% should be filled around each point accorinding to the radius r

n = size(p, 2);
f = zeros(2,n,ceil(pi*(r+1)*(r+1)));
f(:,:,1) = p;
for pt = 1:n
  count = 2;
  for i = 0:r
    for j = 0:r
      if i == 0 && j == 0
        continue;
      elseif i*i + j*j < r*r
        if p(1,pt) + j ≤N
          if p(2,pt) + i ≤N
            f(:,pt,count) = [p(1,pt) + j; p(2,pt) + i];
            count = count + 1;
          end
          if p(2,pt) - i > 0
            f(:,pt,count) = [p(1,pt) + j; p(2,pt) - i];
            count = count + 1;
          end
        end
        if p(1,pt) - j > 0
          if p(2,pt) + i ≤N
            f(:,pt,count) = [p(1,pt) - j; p(2,pt) + i];
            count = count + 1;
          end
          if p(2,pt) - i > 0
            f(:,pt,count) = [p(1,pt) - j; p(2,pt) - i];
            count = count + 1;
          end
        end
      end
    end
  end
end

f(:,:,count:end) = [];
```

A supporting program that creates an RGB image of the simulated particles.

```
function I = fillcolorimage(f, N, color)
% given a set of coordinates f, assign a color to these coordinates in an
% NxN color image

Ir = zeros(N);
Ig = zeros(N);
Ib = zeros(N);
idx = squeeze(f(1,:,:) + N*(f(2,:,:) - 1)); % index into I
M = size(idx, 1);
for i = 1:M
  Ir(idx(i,:)) = color(1,i);
  Ig(idx(i,:)) = color(2,i);
  Ib(idx(i,:)) = color(3,i);
end

% consolidate all channels into one
I(:,:,1) = Ir;
I(:,:,2) = Ig;
I(:,:,3) = Ib;

% add a white border to better see where particles collide
I(1,:,:) = 1;
I(:,1,:) = 1;
I(N,:,:) = 1;
I(:,N,:) = 1;
```

A program that generates the error plots for velocity and position.

```
function ploterr(errp, errv, numneurons, ftrain)
% program to save a plot of position and velocity errors

fig = figure();
ah = axes();
hold on
plot(ah, 1:length(errp), errp);
xlabel(ah, 'Collision');
ylabel(ah, 'Error in position');
title(ah, 'Error in Position after each collision');
set(findall(fig,'type','text'),'fontSize',13,'fontWeight','bold');
set(fig, 'PaperSize', [10.0 5.0]);
set(fig, 'PaperPosition', [0 5 10 5]);
set(fig, 'PaperOrientation', 'landscape');
nn = '';
for i = numneurons'
  nn = [nn '-' num2str(i)];
end
saveas(ah, ['ploterrpsm' nn '_' num2str(ftrain)], 'pdf');
hold off
clf

fig = figure();
ah = axes();
hold on
plot(ah, 1:length(errv), errv);
xlabel(ah, 'Collision');
ylabel(ah, 'Error in velocity');
title(ah, 'Error in Velocity after each collision');
set(findall(fig,'type','text'),'fontSize',13,'fontWeight','bold');
set(fig, 'PaperSize', [10.0 5.0]);
set(fig, 'PaperPosition', [0 5 10 5]);
set(fig, 'PaperOrientation', 'landscape');
nn = '';
for i = numneurons'
  nn = [nn '-' num2str(i)];
end
saveas(ah, ['ploterrvsm' nn '_' num2str(ftrain)], 'pdf');
hold off
```