Final Report for Inverse Dynamics approximation with Neural Networks

Jacob Chen
Computer Science M.Sc
University of British Columbia
Email: jchen114@cs.ubc.ca

Meghana Venkatsway
Computer Science M.Sc
University of British Columbia
Email: meghanav@cs.ubc.ca

***Abstract*—Given the complexities and challenges of modelling biomechanical systems, the approach taken towards the design of such a system becomes an important factor to consider for various applications. While many excellent modelling approaches have evolved over the years, the computational complexity still remains a drawback for many. In our attempt to find a black box model that delivers accuracy of simulating biomechanical models while simultaneously reducing computational complexity, we began looking into neural networks as a possible solution. In this report, we provide our results in attempting to build a neural network that mimics inverse dynamic computation implemented by OpenSim to simulate the right upper arm of a human body.**

I. Introduction

Recently, Neural Networks have started to gain popularity in their usage in Machine Learning applications. They are being applied to fields such as Natural Language Processing, Computer Vision, reinforcement learning, and even animation. This paper will discuss its application to approximating a tool called Inverse Dynamics. The only goal of this paper is to see if we could achieve values close to the Inverse Dynamics Tool shipped with OpenSim. We have selected a musculoskeletal model that we will use for our Neural Network approximation. In order to achieve our goal, we generate random movements in an allowable range in order to train and test our neural network using the Inverse Dynamics tool.

A. OpenSim

OpenSim is a software framework designed to build and share musculoskeletal models, simulate movements, and analyze and visualize those movements using specialized tools. This framework contains, amongst other things, a graphical user interface (GUI) written in java, various built-in musculoskeletal models developed and published by an open source community of researchers and users, a software development kit, APIs and other various tools that are used by researchers or hobbyists for analysis. OpenSim has many open source musculoskeletal models available in their repository. These include human and animal models that can vary in complexity and scope. In our case, we are only looking at the right arm of a human. There are multiple arm models that are available and we have the liberty of choosing one that we believe can achieve our objective simply and effectively.

There are many tools available on the platform that can be used to analyze muscle actuation and joint position. The tool in particular that we are trying to simulate with our Neural Network is the Inverse Dynamics tool. This tool determines the generalized forces at each joint that is responsible for a provided movement.

In our exploration for a suitable upper arm model, we discovered an arm model whose muscles are based on the Hills muscle model. For our project we use OpenSims opensource muscle model, called Dynamic Arm Simulator [1] shown in Figure 1. We use this model as input to the inverse dynamics

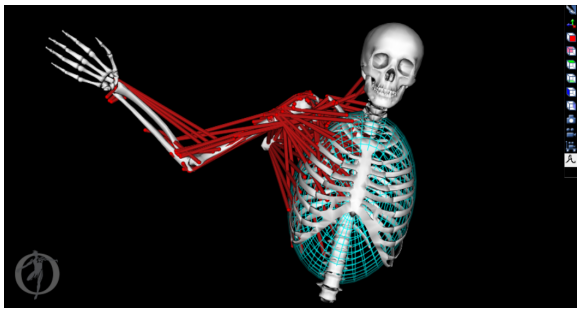

Fig. 1: Open Sim's Arm model

tool to estimate the muscle forces from different movements and postures of the arm. It consists of 11 degrees of freedom, with 31 muscles and 138 muscle elements such as bones, ligaments, etc. and provides a dynamic graphical simulation to visualize and analyze moments about the joints of a human arm.

B. Inverse Dynamics

OpenSims Inverse Dynamics tool receives a motion file, a model, and an optional generalized forces. It then outputs the resulting moments on each joint required for the model to achieve the motion in the prescribed motion file subject to the generalized forces. This process is shown in Figure 2. Thus, it requires a motion file that is transformed from a

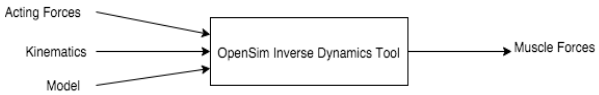

Fig. 2: Open Sim's Arm model

motion capture session into a specified file format dictating the position of each joint in radians. The file specifies the position of the joint with respect to time. Table I illustrates the motion file:

TABLE I: Motion File Table

Time	Joint 0	Joint 1	Joint 2	Joint n
t_0	θ_{00}	θ_{01}	θ_{02}	θ_{0n}
t_1	θ_{10}	θ_{11}	θ_{12}	θ_{1n}
t_2	θ_{20}	θ_{21}	θ_{22}	θ_{2n}
t_m	θ_{m1}	θ_{m2}	θ_{m3}	θ_{mn}

Each of the positions are in units of radians. For our model, we have 11 Degrees of Freedom(joint angles) that correspond to 5 joints,namely sternoclavicular (SC), acromioclavicular (AC), glenohumeral (GH), elbow (EL) and pronation-supination (PS) joints. Each joint has a minimum and maximum value for position expressed as radians. We were not able to find motion capture data online for our model and so we have decided to simulate the motion data programmatically. Our kinematics motion generator takes as input the starting and ending positions of our degrees of freedom and the elapsed time and outputs a motion capture file detailing how each degree of freedom changed for each time step.

II. Background

During this paper, we will be drawing many parallels between our approach and the paper *NeuroAnimator* by Grzeszvzuk [2]. The authors used Neural Networks as a way for reducing the time needed to predict animations on physics based models through numerical methods.

A. NeuroAnimator

Our final project was inspired by the paper *NeuroAnimator* by Grzeszvzuk [2]. In this paper, the authors approximated physical model animation using a neural network that was provided by Xerion, an open platform for neural network usage. The authors were interested to see if given the state at time t (s_t), the control inputs at time t (u_t), and the forces at time t (f_t) of the physical model, if it was possible to produce the next state at $t + \Delta t$ ($s_{t+\Delta t}$). Empirically, their objective was expressed as:

$$s_{t+\delta t} = \Phi[s_t, u_t, f_t] \tag{1}$$

Though Numerical simulation provides unsurpassed realism, predicting each timestep through computation was found to be too time consuming and expensive and so the authors sought out an alternative in the form of a Neural Network. Their goal was to approximate the next state $\Delta t = n\delta t$ where δt is with allowable error while reducing the time needed to predict the next timestep. In the paper, the authors used different physical models of varying complexity to train and test their neural network. At the end they achieved their goal of reducing the amount of time needed for prediction while allowing satisfactory error in the simulations.

III. Neural Network Overview

A Neural Network is loosely based on the computational phenomenon of the brain. Neurons in the brain are connected to each other and fire when the correct stimuli are presented. Inputs are gathered by the Neuron and then pass through an activation function which produces an output from the neuron. These outputs are then propagated throughout the neural network and in the final stage, combined into a output value used to interpret the state of the world. A neuron and network configuration are shown in figure 3.

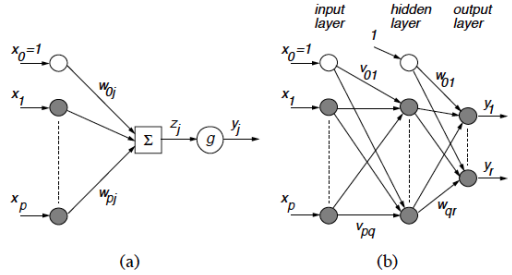

Fig. 3: a) Neuron B) Neural network configuration

Neurons are arranged in layers labeled as input, hidden, or output. Input layers are only responsible for presenting the inputs to the neural network. The hidden layers are sandwiched between the inputs and outputs are do not have any relationship with the outside world. The output layer is the observable. Each neuron in layer l receives weighted inputs from layer $l-1$ and sums them together along with a bias before passing them through an activation function. The bias shifts the input sums along the sigmoid function and can be beneficial to biasing the output towards a particular value. The activation function may change depending on the situation of the neural network, but typically it is the sigmoid function expressed as:

$$g(z) = \frac{1}{1 + e^{-z}} \tag{2}$$

and graphically represented in figure 4.

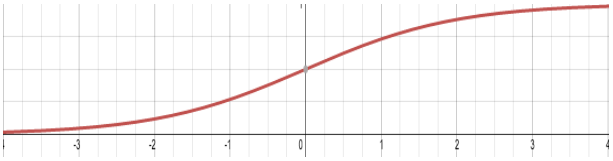

Fig. 4: Sigmoid function a common activation function

Our neural network uses the sigmoid function for all neuron activations. After summing all of the inputs for the neuron, the result is passed through the sigmoid function and the neuron outputs something between 0 and 1 symbolizing the activation. The output is then propagated to the next layer and the process repeats until the final output is reached.

Training a neural network involves modifying the weights and biases associated with each neuron. This is usually done by

taking the gradient of the objective loss function with respect to each parameter. Later we will discuss our loss function and our training algorithm for our specialized neural network.

A. Sampling

As noted previously, the model motion file has 11 degrees of freedom. Each degree of freedom is constrained by an allowable minimum and maximum radian position. To produce our data for training and testing, we first generate random movements that are allowable in the range of motion for the model. The range of motions are found with the model and are shown in figure 5.

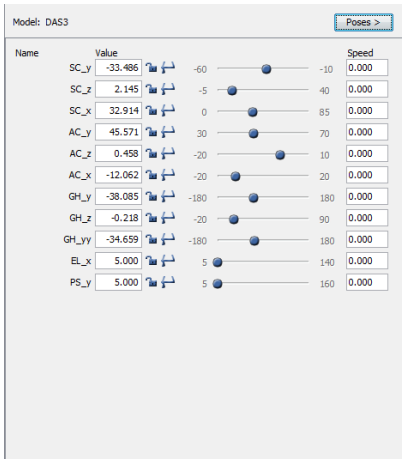

Fig. 5: Allowable ranges for model

Our algorithm for generating random motions is as follows:

for each DOF + time column **do**

1) Generate random start value
2) Generate random end value
3) Interpolate between start and end positions by sampling from position function
4) Record values into motion file

end for

Our position function to sample from is defined as:

$$g(t) = \frac{(50)\left(\frac{1}{k}(x-50)\right)}{\sqrt{1 + \left(\frac{1}{k}(x-50)\right)^2}} + 50 \qquad (3)$$

and graphically represented in figure 6.

The position function represents the position in percentages relative to the starting and final resting positions. The position function value of 0 represents the starting position and 100 represents the final resting position. Timesteps are along the x-axis and each motion file has 100 timesteps representing the trajectory of the motion at each timestep. We expect the motion for each joint to begin slowly, speed up, then slow down as it approaches its final position. The parameters of the position function were determined by what we believed represented a reasonable muscle trajectory when in fact any position function could be used. Once the motion files were

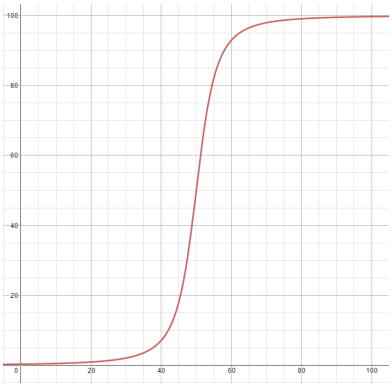

Fig. 6: Position function for sampling

generated, we generated the associated moments file through using the inverse dynamics tool in OpenSim. In order to test our Neural Network, we downsampled the motion files and moments file by a factor $k = 3$. This decreases the amount of neurons used for our network and significantly lowers the amount of time needed for training and testing.

B. Architecture and Construction

For our purposes, we constructed a fully connected multi-dimensional feed forward network with an input layer, 3 hidden layers, and a single output layer. A fully connected feed forward network means that each neuron in the previous layer is connected to all neurons in the next layer. Each hidden layer size is the same as the number of inputs created at the beginning of the network. Our architecture is shown in figure 7.

Each input is a 1×12 vector which represents the state at some

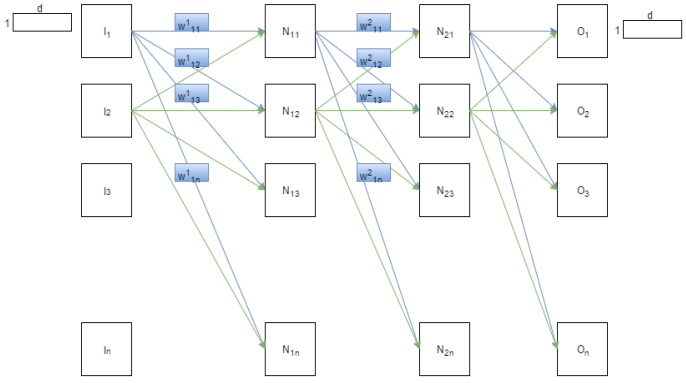

Fig. 7: Neural Network Architecture

timestep. Each weight is multiplied elemental wise with the input and subsequently summed together column-wise. The bias is then added to the result and then each element of the new vector is passed through the sigmoid function. This operation is represented in figure 8.

The number of outputs are the same as the number of inputs and the size for each output and inputs are likewise the same.

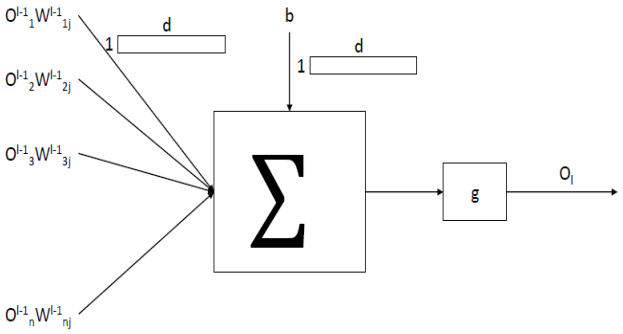

Fig. 8: Neuron Architecture

C. Training

In the paper *NeuroAnimator*, the authors used a series of transformations on the data before training their Neural Network. These transformations are represented in figure 9. The authors have discussed that sanitizing the data presented to the network gave the best results. First, they transformed data from global to local coordinates ($T_x^{'}$), followed by normalization of data to zero mean and unit variance (T_x^{σ}). This expected outputs are also transformed before entering the network through a similar process.

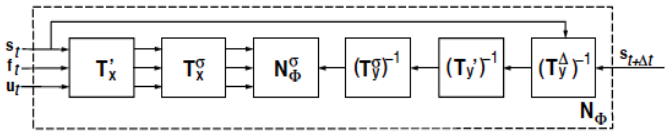

Fig. 9: Neural Network Transformations

For our neural network, we normalized the inputs and outputs by expressing them as values between 0 and 1 defined as:

$$\hat{e_j} = \frac{e_j - min(C_i)}{max(C_i) - min(C_i)} \tag{4}$$

C_i is the i^{th} degrees of freedom. max and min are the maximum values and minimum values of the degree of freedom, and e_j is the j^{th} element in the respective degree of freedom. This transformation is applied to the data and presented to the neural network. There are a few cases where the minimum and maximum value are the same and in those cases we initialize e_j to 0. For training, we transform both the inputs and outputs in this way. This process is illustrated in figure 10.

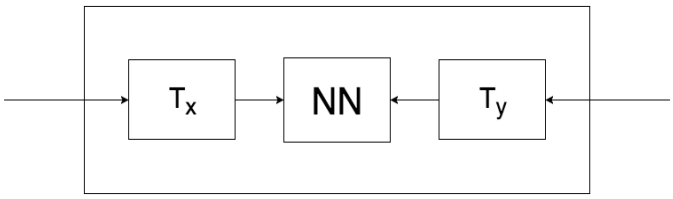

Fig. 10: Neural Network Normalization

We implement our version of training through using a modified back propagation algorithm along with stochastic gradient descent to train our network. Our loss function is expressed as:

$$\arg\min_{w,b} \frac{1}{2} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 \tag{5}$$

where y_i is the associated true output and $\hat{y}_i$ is the predicted output from the neural network. w and b are the weights and biases of the network. We sum across all the outputs of our network to gather all the contributing errors. By finding the gradients with respect to each parameter, we will then update each parameter by taking a step in the direction of the gradient to decrease this loss function.

When computing gradients, we use a method called stochastic gradient descent where a random training example input output pair is selected from the training pool and the gradients calculated from that example are used to update the parameters. This is in contrast to a batch based approach where all the gradients from the training example are averaged and the parameters take a step in that direction. Because of our unique construction of the network where each node receives n 1×12 vectors from the previous layer, each node has an associated $n \times 12$ weight matrix. Each of these elements of the weight matrix will be updated based on the objective loss function. Additionally, each node also has an associated 1×12 bias vector which is added to the sum of the weighted inputs. The individual elements in the bias vector are also updated based on the objective loss function as well.

The algorithm for training is as follows:

Choose a random training example from training pool.
Pass training input through the network and determine objective loss
for Each of the nodes in the output layer **do**
1) Compute gradients with respect to biases
2) Compute gradients with respect to weights
3) Compute gradients with respect to previous layer outputs

end for
{Propagate gradients with respect to outputs backwards through the layers}
repeat
Using output gradients from the next layer:
1) Compute gradients with respect to biases
2) Compute gradients with respect to weights
3) Compute gradients with respect to previous layer outputs

until Input layer is reached
for All weight matrices and bias vectors from nodes **do**
Update all weights and biases based on the computed gradients by a stepsize
end for

For each of the neuron outputs, we know that:

$$O_l^j = g(\sum_{i=1}^{n} (O_i^{l-1} W_{ij}^l) + b_j)$$

Here g represents the sigmoid activation function. O_l^j represents the output of node j in layer l. O_l^j is comprised of the outputs from the previous layer O_i^{l-1}, the weights W of the current layer, as well as the biases b of the current layer. We see that the gradients for the weights, biases, and previous outputs are expressed as:

$$\frac{\partial O_j^l}{\partial W_{ij}^l} = g'(\sum_{i=1}^{n}(O_i^{l-1}W_{ij}^l) + b_j^l)O_i^{l-1}$$
$$\frac{\partial O_j^l}{\partial b_j^l} = g'(\sum_{i=1}^{n}(O_i^{l-1}W_{ij}^l) + b_j^l)$$
$$\frac{\partial O_j^l}{\partial O_i^{l-1}} = g'(\sum_{i=1}^{n}(O_i^{l-1}W_{ij}^l + b_j^l)W_{ij}^l$$

Note that O_i^{l-1} contributes to all of the outputs in the next layer and so in order to back propagate the contribution, we average all of its contributions as follows:

$$\frac{\partial O^l}{\partial O_i^{l-1}} = \frac{1}{N}\sum_{j=1}^{N}(g'(\sum_{i=1}^{n}(O_i^{l-1}W_{ij}^l + b_j^l)W_{ij}^l)$$

Using the above equations we can apply the chain rule to discover the gradients with respect to the weights and biases of previous layers as follows:

$$\frac{\partial O_l^j}{\partial W_{ij}^{l-1}} = \frac{\partial O_l^j}{\partial O_{l-1}}\frac{O_{l-1}}{W_{ij}^{l-1}}$$
$$\frac{\partial O_l^j}{\partial b^{l-1}} = \frac{\partial O_l^j}{\partial O_{l-1}}\frac{O_{l-1}}{b^{l-1}}$$

Once all the gradients have been calculated, we proceed to update each weight element by a step size into the gradient direction. Our step size in our project was 0.5. We used this as a step size because on our runs through training, we saw that the gradients were very small compared to the actual weights. This is probably due to the fact that our normalization required each weight and input to be between 0 and 1 and thus these values will be multiplied together during training to result in a very small value. Therefore we used a step size we thought was large enough to influence the objective error function in each epoch.

D. Base-line testing

In order to test the correctness of our network. We first initialized a random input and output example that would be the same size as our data. In our case, we introduced a random input matrix of size 13×12 and an output matrix of size 13×12. We would like to see if the objective loss function becomes smaller as our network performs more training rounds. Figure 11 shows our results.

Each of the subplots represents a degree of freedom. Here, we initialize our network to have 13 inputs each with 12 dimensions which corresponds to our down sample rate of 3 for 100 time steps. The network has 5 layers where the first and the last are input and output layers, leaving 3 hidden layers. There are 50 training rounds in each of the 50 epochs.

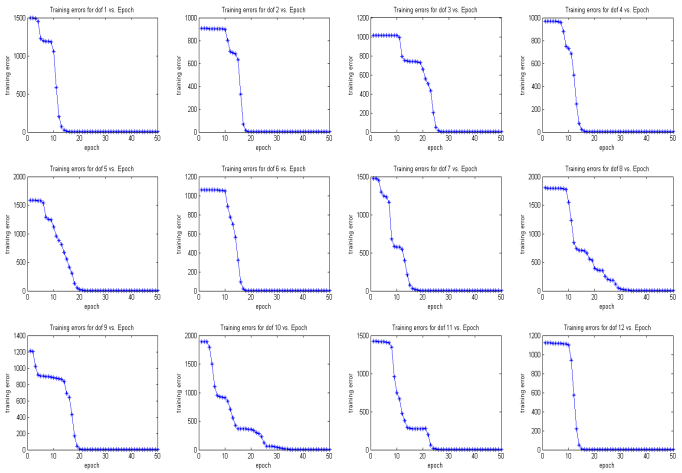

Fig. 11: Baseline testing

In our baseline testing we see that the errors decrease and then level out as the network passes through more epochs, which is what we expect. Note that each of the degrees of freedom behave differently during training.

E. Validation

Once we have verified the behaviour of our network, we begin validation on the data produced from inverse dynamics. Here, we present to our network 2000 samples operating on 2000 training examples for each of the 20 epochs. The network selects 95% of the data for training leaving 5% for validation. The network only updates the weights and biases through training and uses those updated parameters for the validation set. We hope to see that the validation graphs reflects the training error graphs in their behavior. This would indicate that inverse dynamics can be approximated by a neural network. These parameters were chosen experimentally. We looked to see when the training error starting leveling off at a minimum through repeated trials of different parameters. The training error results are shown in figure 12 and the validation error results are shown in 13. Each degree of freedom is shown individually for each graph.

It is interesting to note that the beginning few epochs of training and validation seem to have an error that is constant. After a while it looks like the error drops off rapidly as the network finds a minimum in the objective loss function through the weights and biases of the network. Of particular interest is the first degree of freedom of each graph. This degree of freedom represents the time index. The output times and input times are the same when running the inverse dynamics tool and we see that this error is the lowest error of all the degrees of freedom as would be expected. In our graphs, we see that the validation error mimics the behavior of the training error implying that the computation of the inverse dynamics tool can be approximated using a neural network.

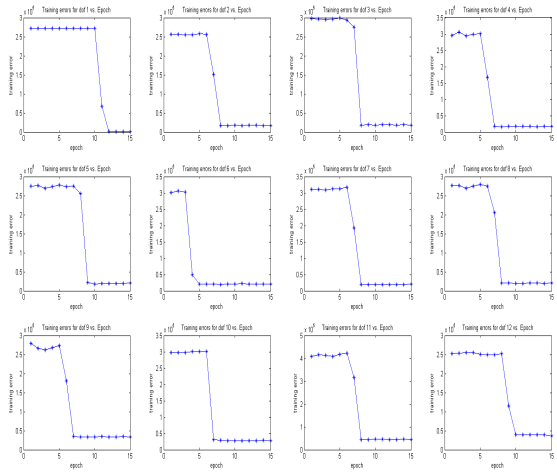

Fig. 12: Inverse Dynamics Training error

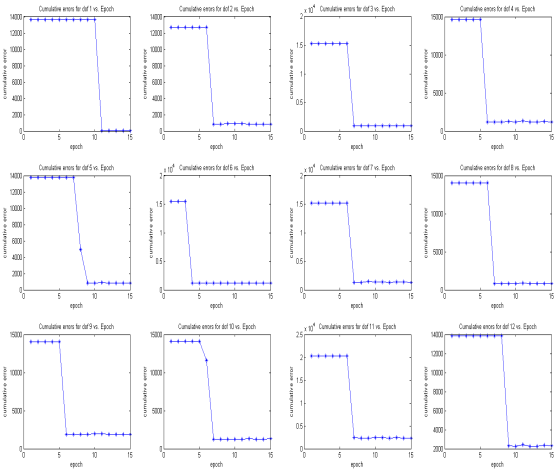

Fig. 13: Inverse Dynamics Validation error

IV. Conclusion

For our project, we set out to build a neural network that could approximate the inverse dynamics computation found in Opensim. We have constructed a unique multidimensional fully connected feed-forward neural network architecture for this task. Our results imply that our attempt of using a neural network for inverse dynamics approximation is successful.

V. Future Work

This neural network is just one of many architectures that can be constructed for this problem. The data for this particular task could be represented in many ways and there is much room for exploration in this regard. Furthermore, we have sampled randomly from the range of motion of the model and different sampling methods could produce varied results. Our normalization step could also be modified to become a different normalization scheme and may be another avenue for exploration.

References

[1] E.K. Chadwick, D. Blana, R.F. Kirsch, and A.J. van den Bogert. Real-time simulation of three-dimensional shoulder girdle and arm dynamics. *Biomedical Engineering, IEEE Transactions on*, 61(7):1947–1956, July 2014.

[2] Radek Grzeszczuk, Demetri Terzopoulos, and Geoffrey 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, pages 9–20, New York, NY, USA, 1998. ACM.