Regression Based Reduced Physics with Control*

Shayan Hoshyari† and Chenxi Liu‡ Department of Computer Science, University of British Columbia

1 Introduction

The relatively low cost of physical simulations has attracted much attention over the past decades. Depending on the application of interest, solution methods to such problems can be considerably different. In the field of computer animation, the usual goal is to obtain visually plausible results in a short amount of time, while engineering communities might be more interested in accuracy at the cost of solution speed.

Machine learning has been a part of many recent contributions in the field of physically based animation. For example, Ladicky et al. [1] developed a random forest based regression framework for free surface fluid simulation and solid interactions. On a set of test cases, they showed a considerable speed up over alternative simulation methods. On a different front, Holden et al. [2] encoded the physics of a game character moving on a terrain via a specially parameterized neural network. They showed that their framework could fairly represent a gigabyte of motion captured data in just a few megabytes. Work has also been done to train sophisticated controllers for accurate physics based on neural networks and reinforcement learning, e.g., [3], which is beyond the scope of this work.

Given the time limit of a course project, we have focused on the physics of a multi-linked pendulum in this work. Following the work of Grzeszczuk et al. [4], we train neural networks that can predict the state of a pendulum after a fixed time-step, Δt, given its initial state and the values of control moments exerted on its links. We then implement a control strategy that can solve for specific control variables over a fixed time range to obtain a desired reduced physics motion.

2 Problem

A dynamical system is usually modelled as a system of differential equations,

$$\dot{s} = f(t, s, u), \tag{1}$$

where s is the vector of state variables, t is time, u is vector of control variables, $\dot{\square}$ shows derivative with respect to time, and the function f encodes the physics of the system. For a three-link pendulum as shown in figure 1-a, the physics of the system is dependent on the mass, M_i, moments of inertia, I_i, and the distance of the center of mass from the top side, LG_i, of the links. The angles of the consecutive links θ_i (figure 1-b) and their rate of change make up the state variables of the system, i.e., $s = [\theta_1, \theta_2, \theta_3, \dot{\theta}_1, \dot{\theta}_2, \dot{\theta}_3]$. The control variables for this problem are the moments τ_i exerted on the the adjacent links as shown in figure 1-c, i.e., $u = [\tau_1, \tau_2, \tau_3]$. Other forces and moments present in the system are gravity and friction, respectively, which are shown in figure 1-d, and would affect the function f. The derivation and the explicit form of the function f is presented in appendix A.1.

*EOSC 550 final project report, April 2018

†M.Sc. student, email: hoshyari@cs.ubc.ca

‡Ph.D. student, auditor, email: chenxil@cs.ubc.ca

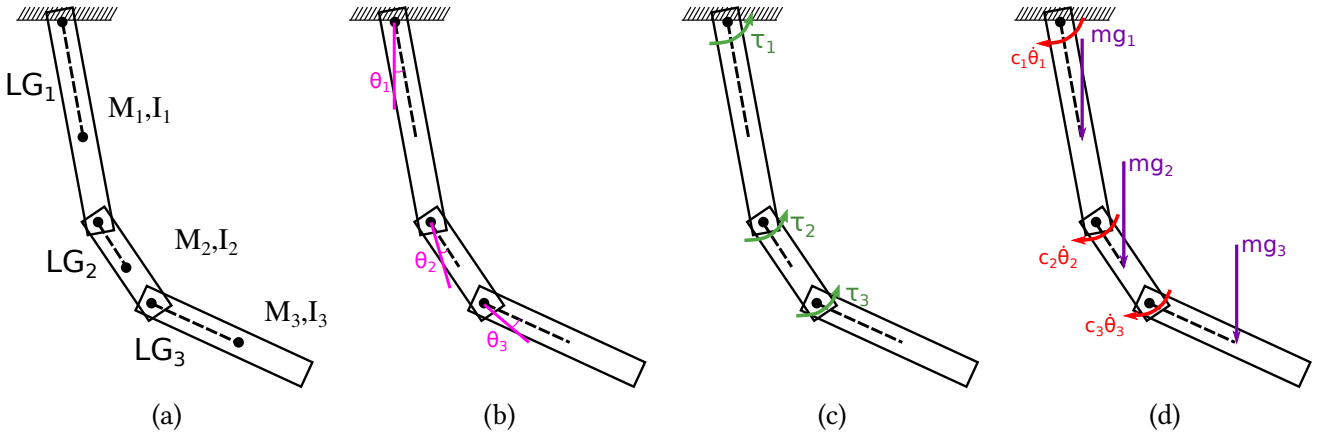

Figure 1: A three-link pendulum: (a) Physical properties (b) degrees of freedom (c) control moments (d) gravity forces and damping moments

We wish to obtain the state function, $s(t)$, given an initial condition s_0, and a control function $u(t) \quad 0 \le t \le \Delta t$. This problem can of course can be solved by integrating equation (1) numerically, e.g., using a Runge-Kutta method,

$$s = \text{RK45}(s_0, u). \tag{2}$$

Here, RK45 represents the adaptive Runge-Kutta-Fehlberg method offered in matlab's `ode45`. Note that the numerical integration method uses an inner time-step usually much smaller than Δt.

Our first goal is to use a series of data generated by this numerically accurate method to train a neural network. Grzeszczuk et al. [4] claim that a well trained neural network can be used as a faster alternative for creating physically based animations of a variaty of systems. Next, we can use the trained neural network to solve a reverse control problem in which one looks for control variables $u(t)$ that produce a desired state function $s_{\text{target}}(t)$:

$$\text{find } u(t) \text{ such that } s_{\text{target}} = \text{NN}(s_0, u).$$

3 Method

3.1 Network Architecture

We use and also compare a single layer and a deeper neural network architecture. Following the notation introduced in the class, the single layer network evaluates the result of regression as

$$C = \sigma(YK + b)W + d,$$

where C is the output of the network, Y is the input, σ is the activation function, K and W are the weights, and b and d are the bias vectors. For the deep network, the state at layer $j+1$, Y_{j+1}, is evaluated as

$$Y_{j+1} = P_{j+1} Y_j + \sigma(Y_j K_j + b_j),$$

where P_j is the identity matrix when dimensions of Y_j and Y_{j+1} match and the zero matrix otherwise. The last Y_j values will then be the output of the network.

Although, we have not performed an extensive search of the network architecture and the hyper-parameters, we have observed that 72 hidden units perform well for the single layer neural network. For the deep network, we have used three inner layers with 20, 20, and 27 hidden units in the first, second, and third layer, respectively. In both cases, the tanh activation function is used.

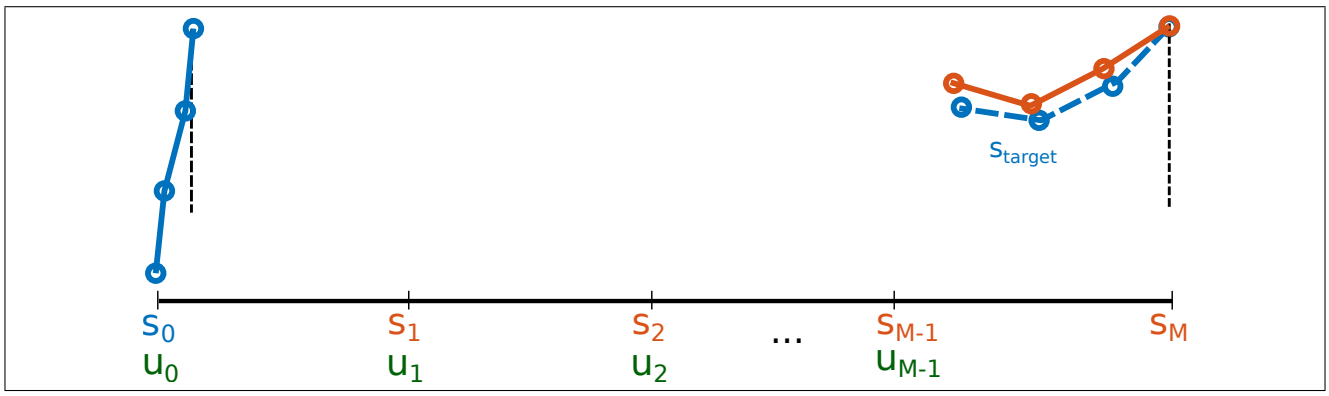

Figure 2: Schematic of the control problem

3.2 Data

We set the feature vector, y, of the neural network to $[s_0, u_0]$, where s_0 is the initial state vector, and u_0 is the values of the control variables. The regression variable is then set to the change in the state vector after one time-step, $C = [s(t = \Delta t) - s_0]$. The hope is that the change in the state vector would have a less complex behaviour compared to its absolute value, and can be reproduced by a neural network more easily (an idea much similar to the residual neural networks). Further, we supply a constant control of $u(t) = u_0$ to the RK45 algorithm when evaluating $s(t = \Delta t)$, as the neural network does not see the changes in u during the $0 \leq t \leq \Delta t$ time interval.

We uniformly sample the training data in a pre-determined range, i.e., the angles, angular velocities, and moments are sampled from the intervals $[-2\pi, 2\pi]$, $[-r_\omega, r_\omega]$, and $[-r_\tau, r_\tau]$, respectively. Here, r_ω and r_τ are hyper-parameters for the data generation process. In the remaining sections we have used $\Delta t = 0.05$, $r_\omega = 2$, and $r_\tau = 10$.

3.3 Optimization

We experimented with a series of optimization methods for training our neural networks, including stochastic gradient descent, Gauss-Newton's method from the minFunc package [5], matlab's `fmin`, and a non-linear conjugate gradient method from the `minimize.m` routine [6]. When using minFunc and matlab's `fmin`, we used the Hessian matrix-vector product of appendix A.2. The `minimize.m` routine had the best performance for our dataset and was used to produce the results that will be presented in section 4.

3.4 Controller

The input to the control problem would be the number of time-steps M, the initial state s_0, and the desired state s_{target}. As shown in figure 2, the goal is to solve for the control variables u_i at time steps $0 \leq i \leq M - 1$, such that applying these controls to a pendulum released from s_0 would produce a motion that will end up as close to s_{target} as possible at time-level M. This problem can be cast into the minimization problem

$$\arg\min_U E(U; s_{\text{target}}, s_0) = \mu \|U\|_2^2 + (s_M - s_{\text{target}})^2,$$

where $U = [u_0, u_1, \cdots, u_{M-1}]$ is the vector of all the controls to be solved for, and μ controls the amount of regularization.

Minimizing the control objective of course requires the derivative of the objective function, which can be subsequently derived by the use of the chain rule. The resulting formula is almost identical to that of backpropagation.

Table 1: Training and test errors for the neural networks

	no control		with control	
Arch.	Train error	Test error	Train error	Test error
deep	4.13%	4.75%	7.23%	7.97%
single layer	7.35%	7.72%	10.57%	11.97%

First, the derivatives of the state at the last time level to those of the previous time levels, $\partial_{s_j} s_M$, are solved for,

$$\begin{bmatrix} I & I+\partial_{s_1} C_2 & & \\ & \ddots & & \\ & I & I+\partial_{s_j} C_{j+1} & \\ & & \ddots & \\ & & & I \end{bmatrix} \begin{bmatrix} \partial_{s_1} s_M \\ \vdots \\ \partial_{s_j} s_M \\ \vdots \\ \partial_{s_M} s_M \end{bmatrix} = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ \vdots \\ I \end{bmatrix},$$

where I is the identity matrix and the matrices $\partial_{s_j} C_{j+1}$ are the derivatives of the output of the neural network with respect to its input state vector. Then, the chain rule can be applied once more to find the derivatives of the final state with respect to the controls at each time level,

$$\partial_{u_j} s_M = \partial_{s_{j+1}} s_M \left(\partial_{u_j} C_{j+1} + I \right),$$

where the matrices $\partial_{s_j} C_{j+1}$ are the derivatives of the output of the neural network with respect to its input control vector. Constructing the the derivative of the objective function, $\partial_{u_j} E$, would then be straightforward by having the $\partial_{u_j} s_M$ values.

4 Results

4.1 Free Motion

We first try to train a neural network to predict the motion of the pendulum without any external control. The following properties of the pendulum and the hyper-parameters for data generation are used.

```
n_links=3; M=[1,1,1]; L=[1,1,1]; LG=L; g=10; I=[0,0,0]; c=[1,1,1]; n_data=5000; r_omega=2pi; r_tau=2;
```

Which result in the convergence history shown in figure 3 during the training process. Training and testing relative L_2 errors are also shown in table 1. We then release the pendulum from the initial condition $s_0 = [2, 0, 0, 0, 0, 0]$, and compare the transient solution obtained from the neural networks and the accurate physical solution, shown in figure 4. The neural networks seem to be capturing the general trend of the solution, albeit with small artifacts. Also, their solution does not seem to decay to zero, and converges to a periodic solution with a small magnitude.

4.2 Forced Motion

We now train the neural networks with the control variables present. The pendulum properties and data generation parameters are this time chosen as:

```
n_links=3; M=[1,1,1]; L=[1,1,1]; LG=L; g=10; I=[0,0,0]; c=[1,1,1]; n=5000; r1=2pi; r2=2
```

The training and testing relative L_2 errors are again shown in table 1. We now release the pendulum from the zero initial conditions while exerting a moment vector of $u = [5{*}\cos(t), 0, \sin(t)]$. The solution of the neural networks

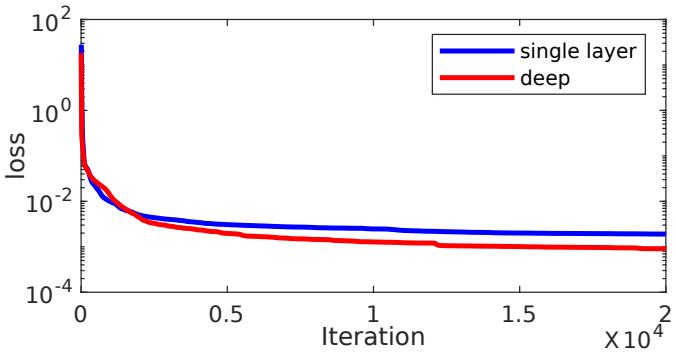

Figure 3: Loss log for training without control variables

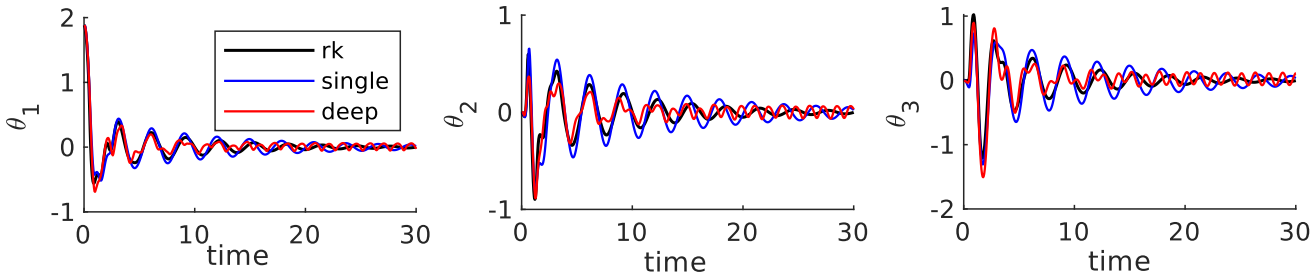

Figure 4: Plot of the solution when the pendulum is released from a non-zero initial state and no control is exerted

and the physical solution are shown in figure 5. The accuracy of the neural network results seem lesser for this case, especially for the second and third angles. Also, the single layer neural network seems to overestimate the solution, while the deep neural network's solution suffers from undershoots.

4.3 Control Problem

In this section, we first train neural networks to predict the physics of a rather simple pendulum:

```
n_links = 3; M = [1 1 1]; L = [1 1 1]; LG = [1, 1, 1]; g = 10; I = [0,0,0]; c = [1, 1, 1].
```

And a more complicated pendulum:

```
n_links = 3; M = [1 1.5 2]; L = [1 1.5 2]; LG = L ./ 2.; g = 10; I = 1/12 .* M .* L .* L; c = [1, 1, 1].
```

Then we use the trained models to control the pendulums released from the position $s = [0, 0, 0, 0, 0, 0]$ to reach the position $s = [-\pi/2, -\pi/4, -\pi/4, 0, 0, 0]$ in 150 time steps. We use a regularization coefficient of 10^{-7}.

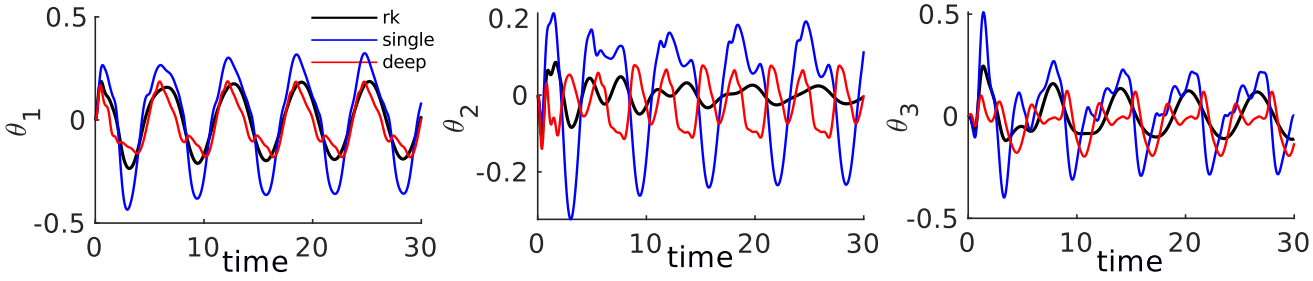

Figure 5: Plot of the solution when the pendulum is released from a zero initial state and oscillatory control values are exerted

For the simple pendulum, the resulting values of angles and moments are shown in figure 6. While both networks successfully reach their target positions, the deep network seems to be producing control moments with less abrupt changes. In other aspects, both architectures seem to be producing similar results. Snapshots of the resulting animation are depicted in figure 7.

For the complicated pendulum, the resulting values of angles and moments are shown in figure 8. None of the architectures can reach the target state in this case. The deep network has a smoother motion, yet more abrupt changes in the moment values. Snapshots of the resulting animation are depicted in figure 9.

5 Conclusion

In this project, we trained a deep and a single layer neural network to encode the physics of a three-linked pendulum. We then used the trained networks for controlling the motion of the system. While we were successful in controlling the motion of a simple pendulum, a complicated pendulum with non-uniform properties proved to be a challenge.

References

[1] L. Ladicky, S. Jeong, B. Solenthaler, M. Pollefeys, M. Gross, et al., Data-driven fluid simulations using regression forests, ACM Transactions on Graphics (TOG) 34 (6) (2015) 199.

[2] D. Holden, T. Komura, J. Saito, Phase-functioned neural networks for character control, ACM Transactions on Graphics (TOG) 36 (4) (2017) 42.

[3] X. B. Peng, G. Berseth, K. Yin, M. Van De Panne, Deeploco: Dynamic locomotion skills using hierarchical deep reinforcement learning, ACM Trans. Graph. 36 (4) (2017) 41:1–41:13. doi:10.1145/3072959.3073602. URL http://doi.acm.org/10.1145/3072959.3073602

[4] R. Grzeszczuk, D. Terzopoulos, 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, 1998, pp. 9–20.

[5] M. Schmidt, minfunc: unconstrained differentiable multivariate optimization in matlab, accessed April 2018. URL http://www.cs.ubc.ca/~schmidtm/Software/minFunc.html

[6] C. E. Rasmussen, Minimize optimization routine, accessed April 2018. URL http://learning.eng.cam.ac.uk/carl/code/minimize/

A Appendices

A.1 Pendulum Governing Equations

There are different ways to obtain the equations of the motion for a pendulum. In this work, we have used the Lagrange equations. The resulting equations would be:

$$M\ddot{\theta} = f(\dot{\theta}, \theta),$$

where M and f are

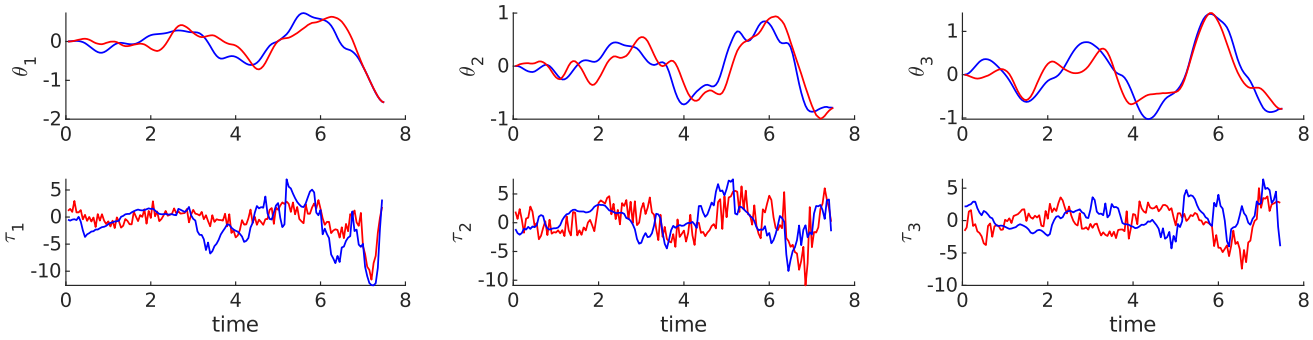

Figure 6: Solution and control moments for the simple pendulum (red: single layer, blue: deep)

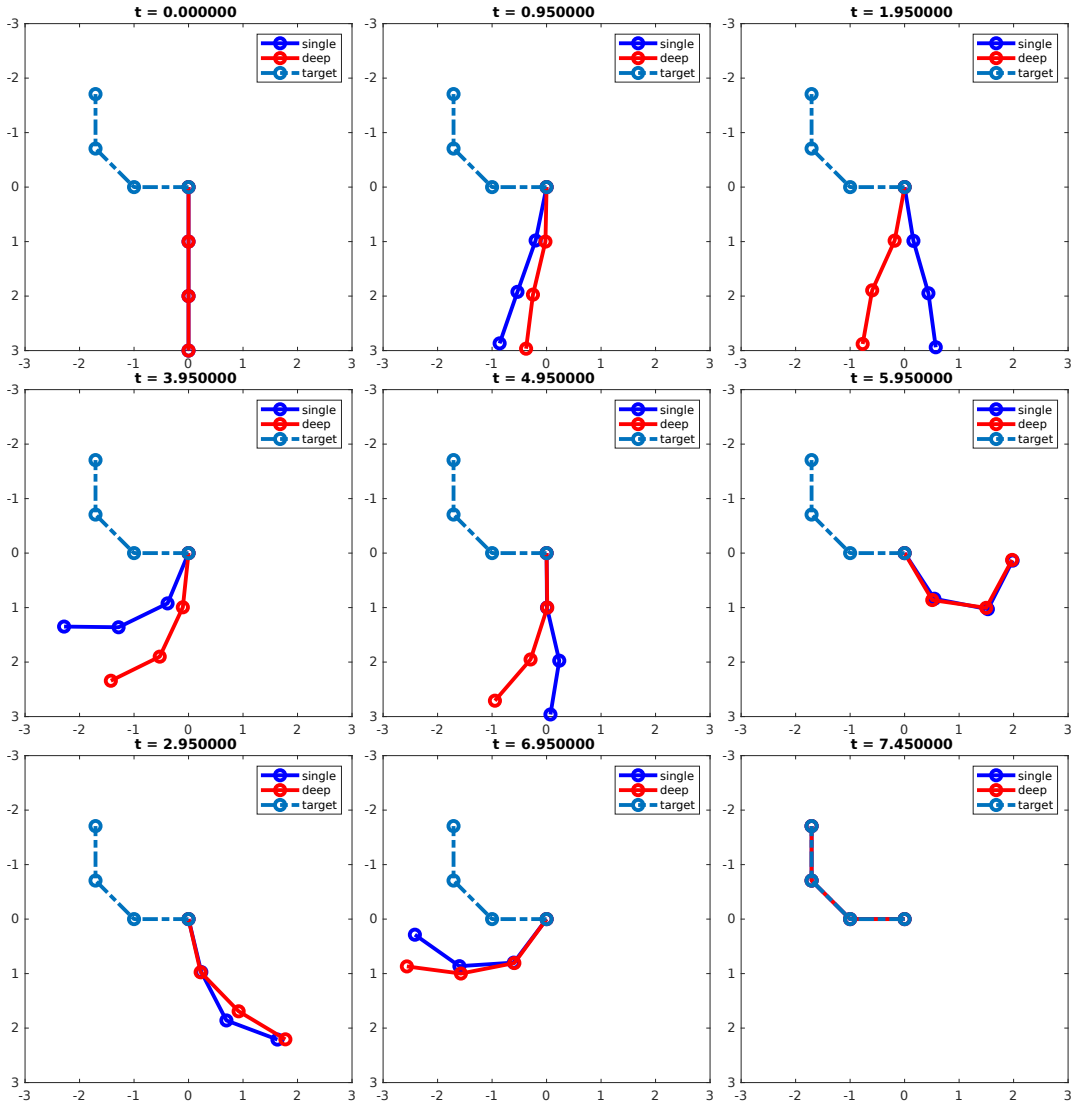

Figure 7: Snapshots of the produced animation for the simple pendulum

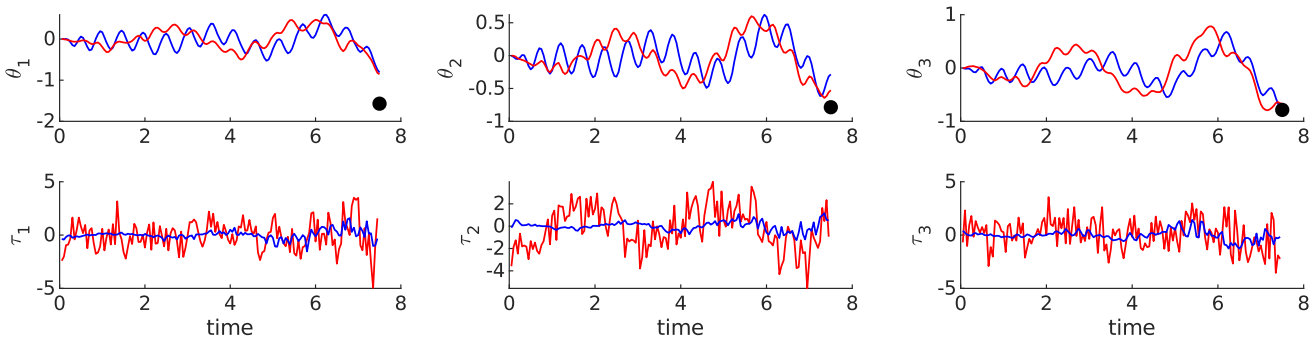

Figure 8: Solution and control moments for the complex pendulum (red: single layer, blue: deep)

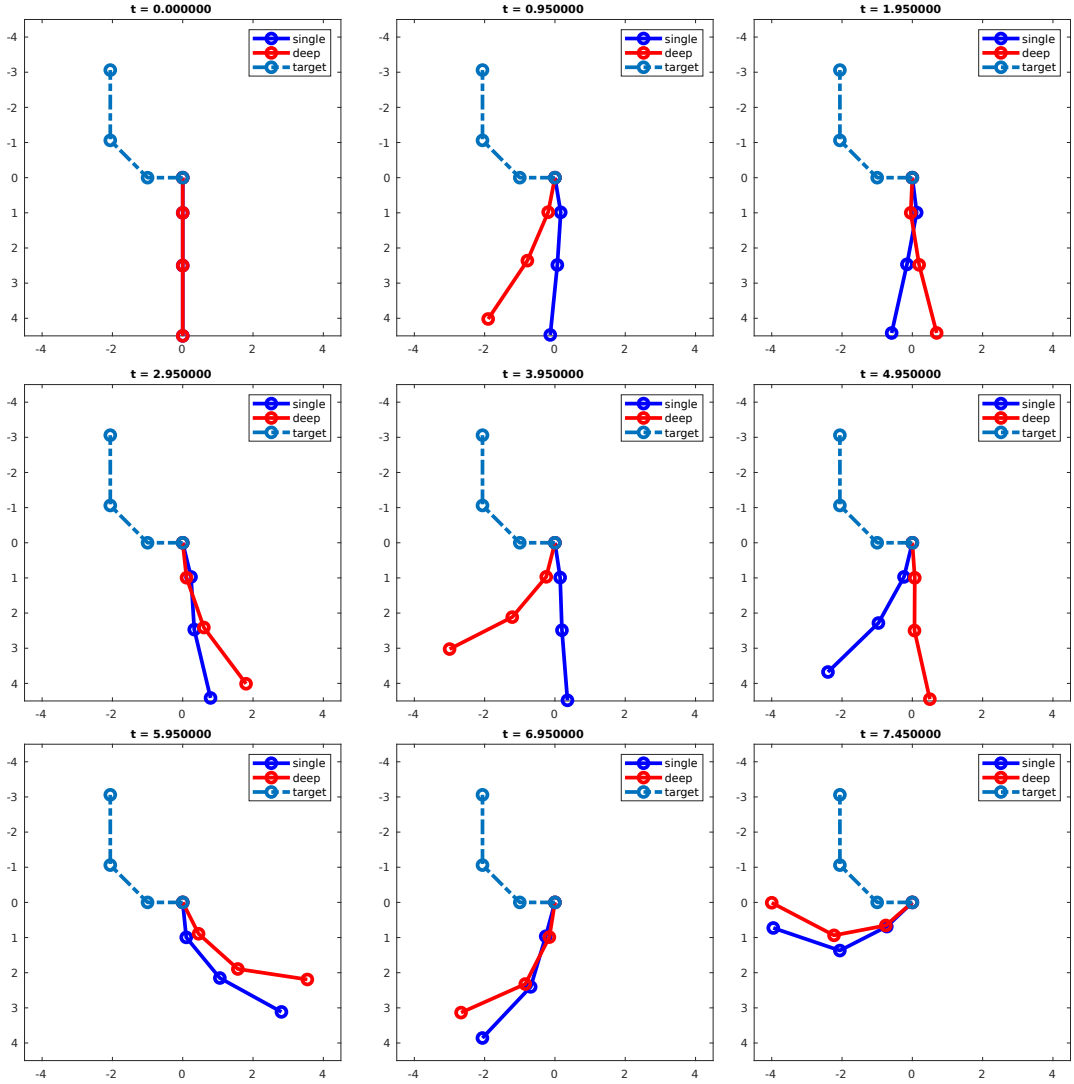

Figure 9: Snapshots of the produced animation for the complex pendulum

```
M(1,1)= m1*LG1*LG1 + (m2+m3)*l1*l1 + I1 + I2 + I3;
M(1,2)= (m2*l1*LG2 + m3*l1*l2)*cos(theta1-theta2) + I2 + I3;
M(1,3)= (m3*l1*LG3)*cos(theta1-theta3) + I3;
%
M(2,1)= (m2*l1*LG2 + m3*l1*l2)*cos(theta2-theta1) + I2 + I3;
M(2,2)= m2*LG2*LG2 + m3*l2*l2 + I2 + I3;
M(2,3)= m3*l2*LG3*cos(theta2-theta3) + I3;
%
M(3,1)= m3*l1*LG3*cos(theta3-theta1) + I3;
M(3,2)= m3*l2*LG3*cos(theta2-theta3) + I3;
M(3,3)= m3*LG3*LG3 + I3;

f(1)= thetadot2*thetadot2 * sin(p2-p1) * (m2*l1*lt2 + m3*l1*l2) + thetadot3*thetadot3 * sin(p3-p1) * (m3*l1*lt3);
f(2)= thetadot1*thetadot1 * sin(p1-p2) * (m2*l1*lt2 + m3*l1*l2) + thetadot3*thetadot3 * sin(p3-p2) * (m3*l2*lt3);
f(3)= thetadot1*thetadot1 * sin(p1-p3) * (m3*l1*lt3           ) + thetadot2*thetadot2 * sin(p2-p3) * (m3*l2*lt3);
%
f(1)= f(1) - g*(m1*lt1+m2*l1+m3*l1)*sin(p1);
f(2)= f(2) - g*(m2*lt2+m3*l2)*sin(p2);
f(3)= f(3) - g*(m3*lt3)*sin(p3);
%
f(1) = f(1) + tau1 - c1*thetadot1;
f(2) = f(2) + tau2 - c2*thetadot2;
f(3) = f(3) + tau3 - c3*thetadot3;
```

A.2 Hessian Matrix-Vector Products for a Single Layer Neural Network

Introducing the notations $Z = \sigma(YK+b)$, $S = ZW+d$, and $E = \frac{1}{2}\|C-S\|_2^2$, while neglecting the second derivative of the activation function, we can derive the following formulas for the Hessian matrix-vector products including derivatives w.r.t K and W:

$$\begin{aligned}
(\partial^2_{KK}E)\delta K = &\quad (\partial_K Z)^T(\partial^2_{ZZ}E)(\partial_K Z)\delta K \\
(\partial^2_{KW}E)\delta W = &\quad (\partial_K Z)^T\left((\partial^2_{ZZ}E)(Z\delta W)W^T + (S-C)\delta W^T\right) \\
(\partial^2_{WK}E)\delta K = &\quad Z^T((\partial_K Z)^T\delta K)W + (\partial_K Z)(\delta K)^T(S-C) \\
(\partial^2_{WW}E)\delta W = &\quad Z^T Z\delta W.
\end{aligned}$$

Derivatives w.r.t b and d can also be derived in a similar manner.