ARL-MR-1121 • MAR 2025

DEVCOM ARMY RESEARCH LABORATORY

Toward Training Physics-Informed Neural Networks without Automatic Differentiation

by Bradley T. Burchett

NOTICES

Disclaimers

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

Citation of manufacturer's or trade names does not constitute an official endorsement or approval of the use thereof.

Destroy this report when it is no longer needed. Do not return it to the originator.

ARL-MR-1121 ● MAR 2025

Toward Training Physics-Informed Neural Networks without Automatic Differentiation

Bradley T. Burchett
DEVCOM Army Research Laboratory

REPORT DOCUMENTATION PAGE

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED	
		START DATE	END DATE
March 2025	Memorandum Report	1 October 2024	19 December 2024

4. TITLE AND SUBTITLE
Toward Training Physics-Informed Neural Networks without Automatic Differentiation

5a. CONTRACT NUMBER	5b. GRANT NUMBER	5c. PROGRAM ELEMENT NUMBER
5d. PROJECT NUMBER	**5e. TASK NUMBER**	**5f. WORK UNIT NUMBER**

6. AUTHOR(S)
Bradley T. Burchett

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	8. PERFORMING ORGANIZATION REPORT NUMBER
DEVCOM Army Research Laboratory ATTN: FCDD-RLA-WD Aberdeen Proving Ground, MD 21005	ARL-MR-1121

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
DISTRIBUTION STATEMENT A. Approved for public release: distribution unlimited.

13. SUPPLEMENTARY NOTES
ORCID: Bradley T. Burchett, 0000-0002-1934-0537

14. ABSTRACT
Artificial neural networks have been used for the past four decades for pattern recognition, system modeling, and feedback control. A recent variant called physics-informed neural network (PINN) incorporates constitutive physical laws into network training to limit solutions to a subspace of physically meaningful models. This report seeks to unpack PINNs by presenting the basic building blocks of shallow networks—neurons, synapses, activation functions, gradients, and backpropagation—and showing some low-dimensional applications thereof. Throughout this work, I use small, shallow networks and analytic gradients to demonstrate fast, smooth solutions without overtraining. I conclude the report with three applications of PINNs: trajectory reconstruction, drag modeling, and optimal control.

15. SUBJECT TERMS
neural network, machine learning, optimal control, parameter estimation, Weapons Sciences

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES
a. REPORT	b. ABSTRACT	C. THIS PAGE		
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UU	37

19a. NAME OF RESPONSIBLE PERSON	19b. PHONE NUMBER *(Include area code)*
Bradley T. Burchett	(410) 306-0792

STANDARD FORM 298 (REV. 5/2020)
Prescribed by ANSI Std. Z39.18

Contents

List of Figures

1. Introduction

Artificial neural networks have been used for pattern recognition, adaptive control, and prediction for the last few decades.[1–3] Recent developments include the introduction of deep neural networks consisting of four or more layers of neurons.[4] More recently, the physics-informed neural network (PINN) came to the forefront for engineering applications. PINNs combine physics-based equations with a standard loss function to constrain the network to a subspace of solutions that are physically meaningful. PINNs have been used to solve nonlinear partial differential equations (PDEs)[5] and estimate system parameters from measured system responses.[6]

Many users have implemented PINNs using deep learning tools such as PyTorch,[7] and automatic differentiation engines such as JAX.[8] Such networks have tens of layers, hundreds of neurons, and thousands of weights.

In this report, the solution to some benchmark PINN problems using minimal networks and analytic differentiation is demonstrated. These methods lead to faster training and eliminate overtraining, resulting in smooth solutions. In Section 2, neural networks and backpropagation without integration are reviewed. Then, networks with integration are reviewed including how to calculate sensitivities when integrators are in the forward path. Then, the dynamic network is made recursive and an example modeling a linear oscillator is shown.

Section 3 shows how static networks can learn a point mass projectile trajectory. Next the static trajectory network is used to find velocity through backpropagation even though the only network output is position. Section 3 concludes by showing how the PINN can be used for drag estimation with noisy measurements.

Section 4 shows an example of using the PINN to solve an optimal control[5] problem. The conditions for optimality are found through the Hamilton–Jacobi–Bellman (HJB) equation. The nonlinear HJB PDE is solved using a PINN and demonstrate a successful feedback control through simulation.

2. Multilayer Neural Networks

2.1 Static Feedforward Networks

Multilayer neural networks consist of several layers which themselves consist of neurons or nodes. The nodes each have an activation function, either linear or nonlinear. Connections (synapses) between nodes have adjustable “weights” that are multiplicative gains. A single node is depicted in Figure 1.

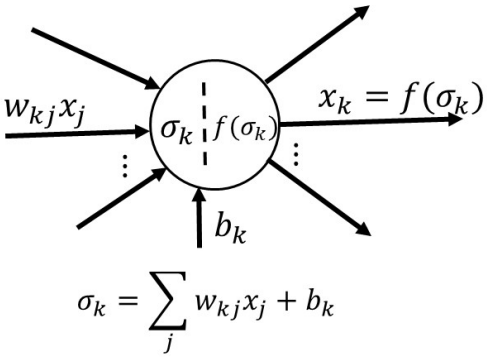


Figure 1. Single hidden neuron with nonlinear activation.

The nonlinear activation function is typically chosen from a set of easily differentiable functions such as log sigmoid:

$$f_1(\sigma) = \frac{1}{1+e^{-\sigma}}, \tag{1}$$

which has the domain, $-\infty < \sigma < \infty$ and the range $0 < f_1 < 1$, or hyperbolic tangent:

$$f_2(\sigma) = \frac{2}{1+e^{-2\sigma}} - 1, \tag{2}$$

which has the domain, $-\infty < \sigma < \infty$ and the range $-1 < f_1 < 1$. These are good choices because their derivatives are readily found as

$$\frac{\partial}{\partial\sigma} f_1(\sigma) = f_1(\sigma)(1 - f_1(\sigma)) \tag{3}$$

and

$$\frac{\partial}{\partial\sigma} f_2(\sigma) = (1 - f_2^2(\sigma)) . \tag{4}$$

A feedforward neural net consists of several layers of nodes as shown in Figure 2.

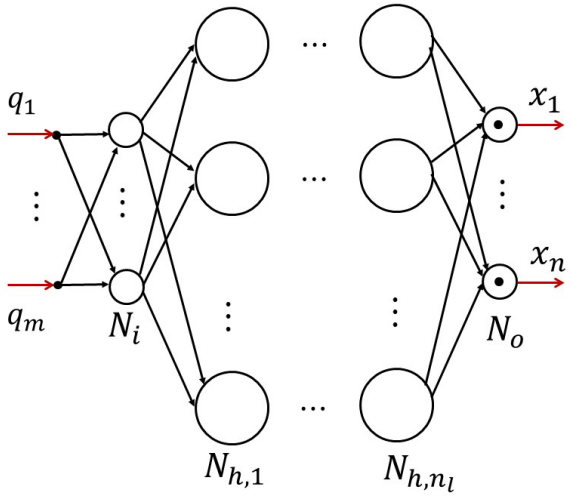


Figure 2. Multilayer feedforward network with n_l hidden layers.

For purposes of this study, the following additional constraints on network architecture are made:

- The input layer has N_i units with $f_2(\sigma)$ activation
- There are n_l hidden layers that all have equal numbers of neurons N_h
- There are N_o output nodes with identity activation $f_o(\sigma_0) = \sigma_o$
- All wires (synapses) are adjustable connecting weights
- All hidden nodes have bias b_k
- Output nodes have no bias
- Input nodes may or may not have bias
- All nodes in layer $l-1$ are connected to all nodes in layer l.

A synapse output is determined by its input from the layer on its left multiplied by its synaptic weight.

$$s_k = w_{kj}x_j \tag{5}$$

Equation 5 relates the three quantities of input, gain, and output. The output of layer k is determined by $x_k = f(\sigma_k)$, where σ_k can be written as a sum over the synapse values fed from the previous layer.

$$\sigma_k = \sum_{N_j} w_{kj}x_j + b_k \tag{6}$$

Equation 6 can be written in matrix-vector form as follows:

$$\boldsymbol{\sigma}_l = \begin{bmatrix}\mathbf{w}_{l,l-1}^T & \mathbf{b}_l\end{bmatrix} \cdot \begin{Bmatrix}\mathbf{x}_{l-1} \\ 1\end{Bmatrix}. \tag{7}$$

Thus, for multilayer feedforward nets, Equations 2 and 7 are applied repeatedly

$$\hat{\mathbf{y}} = \mathbf{x}_o = \mathbf{f}_o(\mathbf{w}_o \cdot \mathbf{f}_{o-1}(\mathbf{w}_{o-1}^T \cdot \mathbf{f}_{\boldsymbol{i}}(\cdots) + \mathbf{b}_{o-1}))$$

and for one hidden layer it is

$$\hat{\mathbf{y}} = \mathbf{x}_o = \mathbf{f}_o\Big(\mathbf{w}_o \cdot \mathbf{f}_h(\mathbf{w}_h^T \cdot \mathbf{f}_{\boldsymbol{i}}(\mathbf{w}_h^T \cdot \mathbf{x}_{\boldsymbol{i}} + \mathbf{b}_{\boldsymbol{i}}) + \mathbf{b}_h)\Big). \tag{8}$$

Multilayer nets are trained through supervised learning, typically using the current error and gradient to determine weight updates. If a sum square error loss function is used,

$$\boldsymbol{E} = \frac{1}{2}\sum_{p=1}^{N_{train}}(y_p - \hat{y}_p)^2,$$

the residual vector is defined as: $\mathbf{R} = \mathbf{y} - \hat{\mathbf{y}}$ for all training pairs *p*, then $\boldsymbol{E} = \mathbf{R}^{\boldsymbol{T}}\mathbf{R}/2$. To train the weights, the gradient of the cost with respect to the weights ($\partial \boldsymbol{E}/\partial \mathbf{w}$) is sought. This is done by repeated application of the chain rule. Starting from the output layer, the gradient of the loss function with respect to a prediction $\hat{y}_p$ is

$$\frac{\partial E_p}{\partial \hat{y}_p} = -\big(y_p - \hat{y}_p\big) = -\delta_o^p \tag{9}$$

$$\frac{\partial E_p}{\partial w_{kj}} = \frac{\partial E_p}{\partial \hat{y}_p} \cdot \frac{\partial \hat{y}_p}{\partial \sigma_k} \cdot \frac{\partial \sigma_k}{\partial w_{kj}} \tag{10}$$

and Equation 10 is the differentiation chain from the output to a weight w_{kj} connecting the penultimate layer to the output layer, where

$$\frac{\partial \sigma_k}{\partial w_{kj}} = u_j; \quad \frac{\partial \hat{y}_p}{\partial \sigma_k} = f'(\sigma_k)$$

$f'(\sigma) = 1$ for identity activation, and ${f_1}'(\sigma)$, and ${f_2}'(\sigma)$ are given by Equations 3 and 4, respectively, for nonlinear activation.

Thus, for an output unit ($f'(\sigma) = 1$)

$$\frac{\partial E_p}{\partial w_{kj}} = -\big(y_p - \hat{y}_p\big)u_j^p \ . \tag{11}$$

For hidden units, the error of the downstream layer is *backpropagated*. In other words, to capture the influence of a hidden unit on all units in the next layer, the δs of all units in the next layer must be summed by multiplying by the connecting weights. If $\boldsymbol{\sigma}_l = \begin{bmatrix}\mathbf{w}_{l,l-1}^T & \mathbf{b}_l\end{bmatrix} \cdot \begin{Bmatrix}\mathbf{x}_{l-1} \\ 1\end{Bmatrix}$, then

$$\delta_j = f'(\sigma_j^p) \sum_{Nk} \delta_k^p w_{kj} \tag{12}$$

or $$\delta_j = f'\left(\sigma_j^p\right)\mathbf{w}_{kj}\boldsymbol{\delta}_k^p = \partial E_p/\partial\sigma_j \ .$$

To determine all weight sensitivities, Equation 12 is applied recursively backward through the network. The sensitivity with respect to the weight is found by

$$\frac{\partial E_p}{\partial w_{kj}} = \frac{\partial E_p}{\partial \sigma_k} \cdot \frac{\partial \sigma_k}{\partial w_{kj}} = \delta_k x_j \ . \tag{13}$$

In this work, the Levenberg–Marquardt (LM) training algorithm is used. This is a quasi-Newton method based on finding the normal equation $\mathbf{R} = \mathbf{J} \cdot \boldsymbol{\delta}\mathbf{w}$ and solving for weight updates by a modified least squares

$$\boldsymbol{\delta}\mathbf{w} = (\mathbf{J}^T\mathbf{J} + \mu\mathbf{I})^{-1}\mathbf{J}^T\mathbf{R} \ , \tag{14}$$

where **J** is a Jacobian matrix formed with rows corresponding to individual training errors and columns corresponding to individual weights, $\mathbf{J}_{ij} = \partial e_i/\partial w_j$, $\mathbf{R}_i = e_i$. For vector valued outputs, the rows are interleaved between the various outputs. μ is a small positive constant used to ensure good conditioning of Equation 14 and is adjusted according to the success or failure of a trial weight update. The LM proceeds as follows:[9]

1) Forward propagate to predict measurement $\widehat{\mathbf{Y}}$
2) Backpropagate to compute sensitivities using Equation 12, with $\delta_o = 1$.
3) Form Jacobian from all sensitivities as $\mathbf{J}_{ij} = \partial e_i/\partial w_j$ and form residual as $\mathbf{R}_i = e_i$ with rows corresponding to those of **J.**
4) Make a trial correction $\mathbf{W}^{i+1} = \mathbf{W}^i - (\mathbf{J}^T\mathbf{J} + \mu\mathbf{I})^{-1}\mathbf{J}^T\mathbf{R}$.
5) Find the updated residual by repeating forward propagation with new weights $\mathbf{W}^{i+1}$.
6) If the error $\|\mathbf{R}\|_2$ is reduced, set $\mu = \mu/\beta$ and repeat from Step 1. If not, set $\mu = \mu\beta$ and repeat from Step 4.

2.2 Dynamic Feedforward Nets

To add the capability to model dynamic phenomena (systems where the history of the system impacts its future behavior), an integrator is applied to the output of each hidden layer unit (Figure 3).

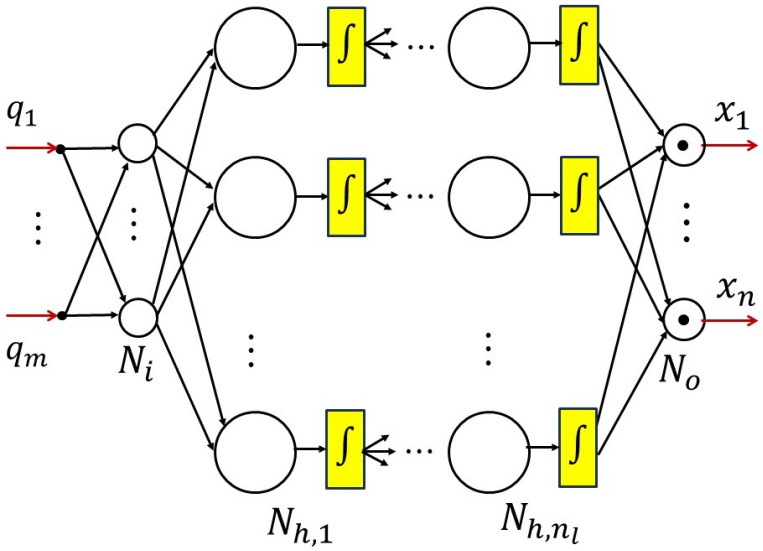


Figure 3. Dynamic feedforward neural network.

Such that,

$$x_k = \int f\left(w_{kj}x_j + b_k\right)dt \; . \tag{15}$$

Since the variable of integration, *t*, is independent of the weights, backpropagation may be performed exactly as before except that the result will be

$$\delta_k x_j = \frac{d}{dt}\left(\frac{\partial e_p}{\partial w_{kj}}\right) \tag{16}$$

for units in the ultimate hidden layer, and

$$\delta_k x_j = \frac{d^2}{dt^2}\left(\frac{\partial e_p}{\partial w_{kj}}\right) \tag{17}$$

for those in the penultimate layer, and so on. Thus, sensitivities of weights to the left of integrators will need to be integrated a requisite number of times after backpropagation. In continuous time networks, this integration should be performed simultaneously with the integrations in the forward path. In other words, the right-hand sides of Equations 16 and 17 are treated as co-states during the forward net integration.[10,11]

2.3 Recursive Dynamic Neural Nets

Dynamical systems are often modeled with recursive ordinary differential equations. That is, the state derivatives are dependent upon the states as well as time and inputs, for example, $\dot{\mathbf{x}} = \mathbf{f}(t, \mathbf{x}, \mathbf{u})$. For a network to reflect this type of dynamics, system states need to be fed back as inputs. Figure 4 shows an

architecture that has the potential to model many dynamical systems. In this formulation, the output of every integrator is fed back to the input layer. This resembles the control canonical form of the simulation diagram for linear systems.[12]

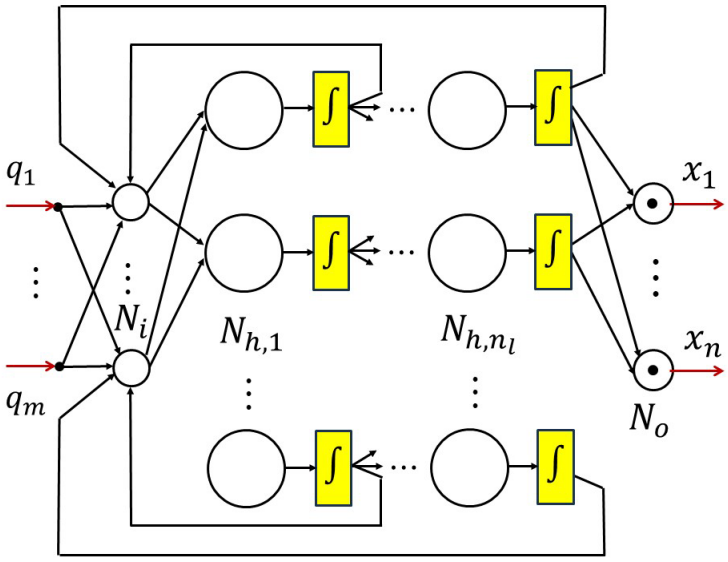


Figure 4. Recursive dynamic neural network.

To properly handle recursion, forward- and backpropagation must be performed at each time step to determine the inputs to system integrators, and co-states as shown in Equations 16 and 17. All state derivatives are combined into a single-state vector and integrated forward in time. More specifically, the current state of all integrators in the network is fed back and concatenated to the input vector.

Forward propagation is performed, skipping the integrators and resuming at integrator outputs. Backpropagation is done in customary fashion, ignoring the integrators. Sensitivities of input layer weights will be found as a product of input layer deltas and all inputs, including the fed-back states (Equation 13). Since the input layer is left of N_l hidden layers, these sensitivities will need to be integrated l times (Equations 16 and 1).

A dynamic recursive network modeling the two-state system is demonstrated as follows,

$$\dot{x}_1 = x_2 - t \tag{18}$$

$$\dot{x}_2 = -16x_1 + 16\sin(4\pi t),$$

which is an undamped oscillator with harmonic forcing. Only x_1 is measured in this example. For this problem, a dynamic recursive net with two hidden layers of seven units each is formed. All 14 integrators will be fed back to the input layer. This

architecture is chosen as the smallest two hidden layer network that will provide satisfactory results. Two hidden layers make intuitive sense since the system being modeled has two integrators. The input is assumed to be $u = \sin(4\pi t)$. The input layer has bias, bringing the total number of inputs to $2 + n_l \cdot n_h = 16$. Thus, $\mathbf{w_i}$ is 16×7 and each hidden layer has an 8×7 matrix of connecting weights. The output gain matrix is 1×7 for a grand total of 231 adjustable weights in the network.

The network was trained for 5000 epochs using LM. Each epoch consists of the following steps:

- Integrate from t_0 to t_f
- At each time step,
    - Forward- and backpropagate to find integrator states and sensitivity states
- Form the Jacobian from all sensitivities (all but $\partial e_i / \partial w_o$ are determined by co-state integration)
- Make a trial update
- Use μ iteration until error is reduced, or μ exceeds $\|\mathbf{J}\|_2$

Figure 5 shows the convergence of network prediction error over 5000 epochs. Sum squared error is reduced to 0.051 m^2 and could be reduced further through additional epochs.

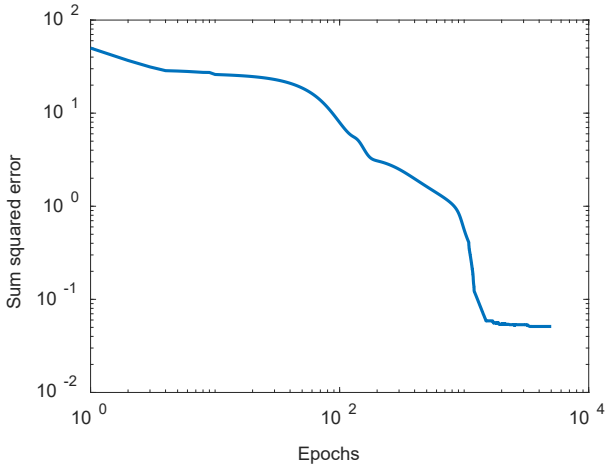


Figure 5. Convergence of training the recursive dynamic neural network.

Figure 6 shows the target system trajectory for x_1 and the network prediction after training. The prediction accuracy could be improved through further training; however, this result confirms that this network architecture is appropriate for modeling $\dot{\mathbf{x}} = \mathbf{f}(t, \mathbf{x}, \mathbf{u})$.

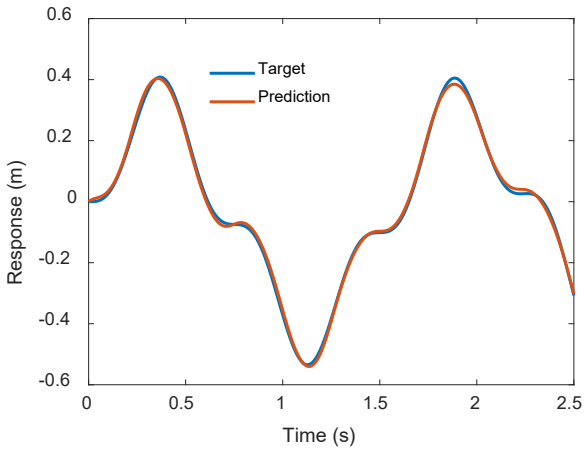


Figure 6. Undamped oscillator time response and network prediction after training.

3. Modeling Trajectories with Static Feedforward Networks

One approach to parametric identification is to first model a system trajectory with a static feedforward network with time as the sole input.[6] In this section, the example of modeling a point mass trajectory with time-dependent drag is explored. The true system dynamics are as follows:

$$\begin{aligned} \dot{x} &= V_x \\ \dot{y} &= V_y \\ \dot{V}_x &= -C_D^* V_\infty V_x \\ \dot{V}_y &= -C_D^* V_\infty V_y - g \ , \end{aligned} \tag{19}$$

where $C_D^* = \rho A C_D / 2m$, $C_D = \sin(2\pi t) + \frac{1}{2}$, and $V_\infty = \sqrt{V_x^2 + V_y^2}$. Equation 19 models a point mass flying through the atmosphere with quadratic drag. The drag coefficient could vary harmonically if the body were tumbling in flight such that the profile drag varied accordingly.

Training data is generated by integrating Equation 19 from $t = 0$ to $t = 2.5$ s with $V_\infty(0) = 60$ m/s, $V_x(0) = V_\infty \cos 10°$, and $V_y(0) = V_\infty \sin 10°$. $M = 1.146$ kg, $A = 0.0038$ m^2, and $\rho = 1.225$ kg/m^3. The trajectory was then modeled by a feedforward network with one input, two outputs, and two hidden layers with seven units each. The training data is scaled such that $-1 \le \tilde{x} \le 1$ and $-1.5 \le \tilde{y} \le 0.5$. Scaling the data to approximately unity at both input and output benefits training for two reasons. First, the activation functions $f_1(\sigma)$ and $f_2(\sigma)$ have approximately 80% of their variation for inputs $-1 \le \sigma \le 1$. Thus, the network is more capable of learning nonlinear mappings with inputs in this range. Second, the activation

functions have range around unity scale. If the modeled output is much larger than unity, the output weights must be large, resulting in large sensitivities during training, and internal signals "firewalling," such that $f(\sigma) \to 1$ or $\to -1$ for some units. Velocities were scaled corresponding to their position counterparts: $\tilde{V}_x/V_x = \tilde{x}/x, \tilde{V}_y/V_y = \tilde{y}/y$. The network was trained for 1070 epochs to match the trajectory $(x(t), y(t))$. Figure 7 shows the sum squared error as the network converges to the training data. After 1070 epochs, the sum squared error is below 10^{-6}.

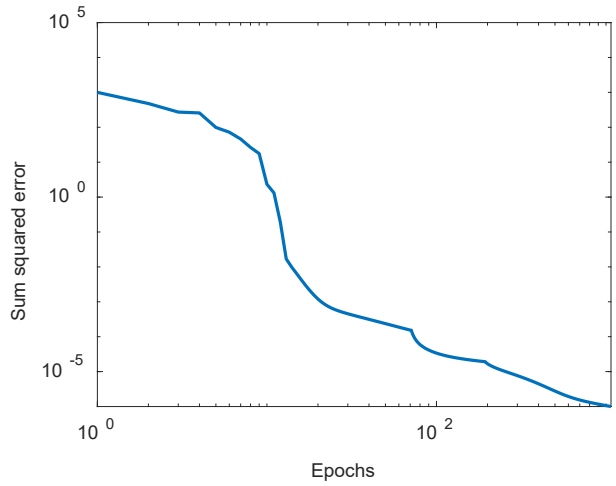


Figure 7. Convergence of training the one input trajectory prediction network.

Figure 8 compares the network prediction with the true trajectory after training and rescaling.

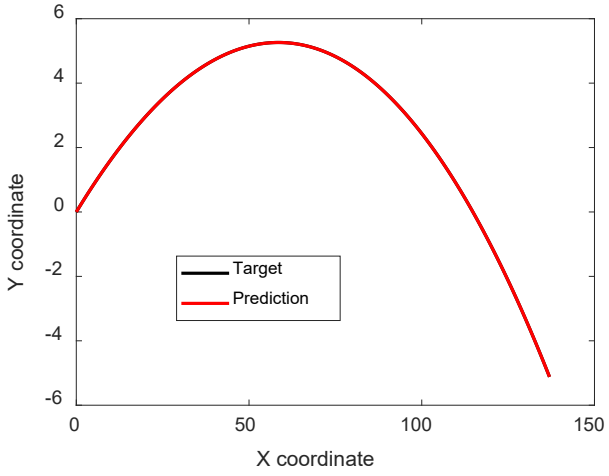


Figure 8. Comparing actual trajectory and one input network prediction.

3.1 Finding Vx(t), Vy(t) as Derivatives with Respect to the Input

With a two-output system, each layer will have two δ values. Applying Equation 12, sensitivities of the ultimate hidden layer weights are computed as

$$\delta_l^1 = f'(\sigma_l^p) \sum_{Nl} \delta_o^p w_{1l}$$

i.e., $f'\left(\sigma_j^p\right) \cdot w_{1l}^T$ where w_{1l}^T is the first row of the output matrix and

$$\delta_l^2 = f'\left(\sigma_l^p\right) \sum_{Nl} \delta_o^p w_{2l} = f'\left(\sigma_l^p\right) \cdot w_{2l}$$

Each of these deltas must be backpropagated through the network. Signal relationships at the input layer are depicted in Fig 9.

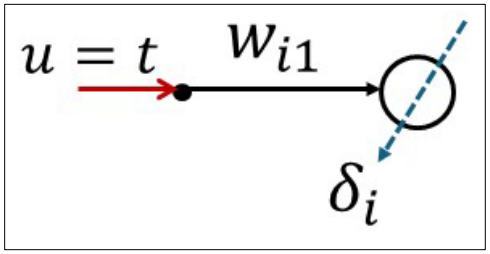


Figure 9. Signal relationships at an input unit.

Figure 9 indicates that $s_{i1} = w_{i1}t$ in forward propagation and $\partial e^p / \partial w_{i1} = \delta_i^1 t$. Since the output is always $f(w_{i1}u)$ regardless of number of layers, consider the following relationships

$$\frac{\partial f}{\partial w_{i1}} = f'(\cdot)u = \delta_i^1 u \tag{20}$$

and

$$\frac{\partial f}{\partial u} = f'(\cdot)w_{i1} \, . \tag{21}$$

Comparing Equations 20 and 21, it is deduced that

$$\frac{\partial f}{\partial u} = f'(\cdot)w_{i1} = \delta_i^1 w_{i1} \tag{22}$$

or when the input is time $\frac{\partial f}{\partial t} = \delta_i^1 w_{i1}$. Thus, the network predictions of $\frac{\partial x}{\partial t} = V_x$ and $\frac{\partial y}{\partial t} = V_y$ are found without automatic differentiation. This will come in handy

when it is time to solve PDEs with PINNs where the PDE consists of partial derivatives with respect to the inputs.

Using Equation 22 for the network trained on position data only, I predicted the vertical and horizontal velocities. These predictions are compared to the truth data in Figure 10, where the true velocity is plotted as a thick blue line and network prediction is shown in red. The network prediction is good for most of the trajectory but differs significantly at the beginning ($V_x = 59.1$ m/s) and end ($V_x = 50.9$ m/s).

This velocity prediction was determined to be inadequate for drag estimation using a PINN. Thus, prior to training the PINN, training the network using both position (network output) and velocity (from backpropagation) as targets is explored.

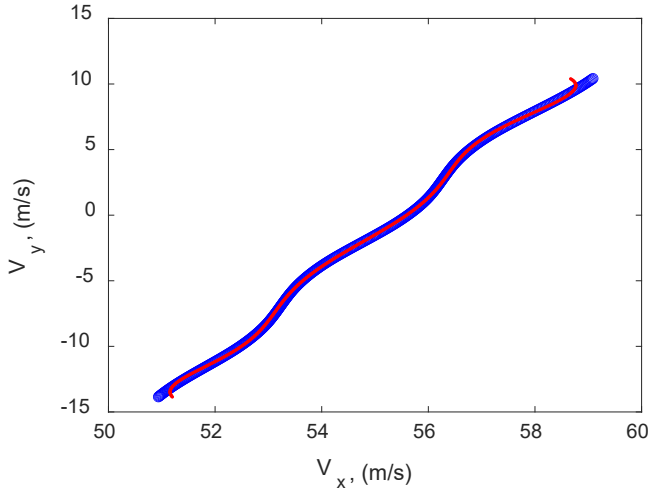


Figure 10. Velocity prediction from a network trained on position data only.

3.2 Introducing Velocity Feedback

To refine the network prediction of velocity as determined by Equation 22, the number of network outputs is fixed at two, but Equation 22 is employed to predict V_x and V_y. These predictions are interleaved with the position predictions. Velocity measurements are likewise interleaved with position measurements. The residual vector is found as the difference of measurements and predictions.

Next, the **J** matrix must be appended with rows consisting of the sensitivities of the velocity predictions with respect to the weights. The easy way to do this is to realize these rows are defined as

$$J_{V_x} = \frac{\partial}{\partial w_{kj}}\left(\frac{d}{dt}x\right) \tag{23}$$

$$J_{V_y} = \frac{\partial}{\partial w_{kj}}\left(\frac{d}{dt}y\right) \tag{24}$$

and simply reverse the order of differentiation

$$J_{V_x} = \frac{d}{dt}\left(\frac{\partial x}{\partial w_{kj}}\right) \tag{25}$$

$$J_{V_y} = \frac{d}{dt}\left(\frac{\partial y}{\partial w_{kj}}\right). \tag{26}$$

Since t is the independent variable along the vertical dimension of **J** and $\frac{\partial x}{\partial w_{kj}}$, $\frac{\partial y}{\partial w_{kj}}$ are elements of the existing **J**, finite differencing the existing columns of **J** to approximate J_{V_x} and J_{V_y} results in

$$\frac{d}{dt}\left(\frac{\partial x}{\partial w_{kj}}\right) = \frac{\Delta\left(\frac{\partial x}{\partial w_{kj}}\right)}{\Delta \mathrm{t}}$$

and

$$\frac{d}{dt}\left(\frac{\partial y}{\partial w_{kj}}\right) = \frac{\Delta\left(\frac{\partial y}{\partial w_{kj}}\right)}{\Delta \mathrm{t}}$$

Training was repeated on the previous network with targets consisting of position and velocity measurements and **J** appended as described above. The training required 1600 epochs to reach a sum squared error of $1.53(10^{-4})$. Figure 11 shows the convergence of the network during this training.

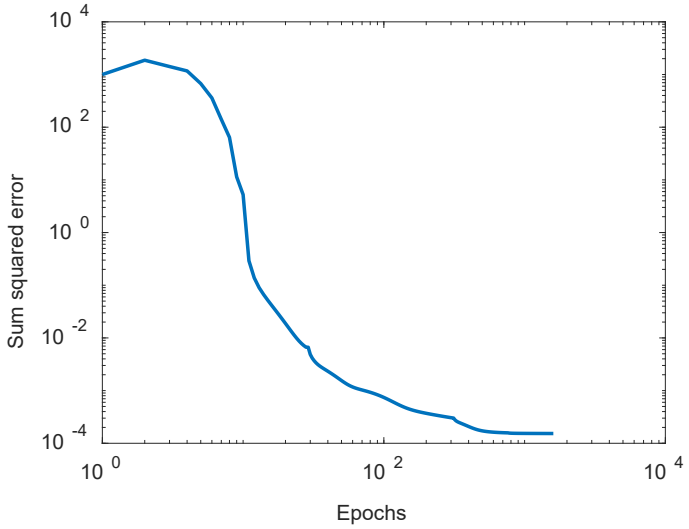
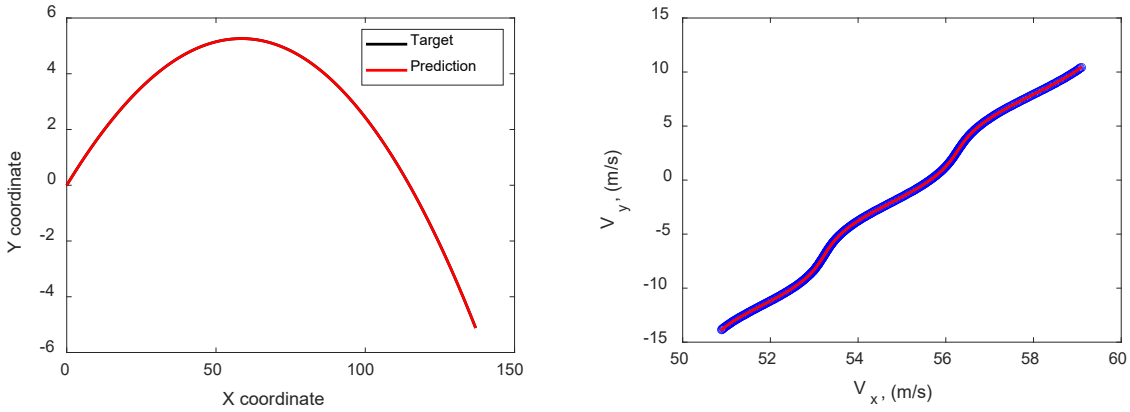


Figure 11. Trajectory matching training using position and velocity measurements.

The resulting trajectory predictions are compared to training data in Figure 12. Velocity prediction is now accurate enough to proceed with drag prediction through a PINN.

The trajectory matching was repeated with velocity feedback for a 3-D point mass flight as well. The results are shown in the Appendix.



a) Position prediction compared to truth

b) Velocity estimates compared to truth with additive noise

Figure 12. Position and velocity predictions compared to actual after training.

3.3 Atmospheric Drag Estimation PINN

A PINN is a network that is set up and trained to satisfy constitutive relationships from physics. This restricts solutions rendered by neural nets to ones that are physically meaningful. "Set up" means the network architecture provides some signals that are physically meaningful such as velocity feedback found in the previous section. Training is determined by the loss function. For instance, if a loss function is modified to include a term that measures how close the network solution is to satisfying physical laws, we have a PINN. In the simplest case, we might want a network prediction $\hat{\boldsymbol{y}}$ to match training data $\boldsymbol{y}$ while also conforming to Newton's second law $\boldsymbol{F} = \boldsymbol{m\ddot{y}}$. A loss function would be set up that looked like

$$\boldsymbol{J} = \boldsymbol{\alpha} \sum_{N_p} \left(\boldsymbol{y_p} - \hat{\boldsymbol{y}}_{\boldsymbol{p}}\right)^2 + \boldsymbol{\beta} \sum_{N_p} \left(\boldsymbol{F_p} - \boldsymbol{m\ddot{y}_p}\right)^2,$$

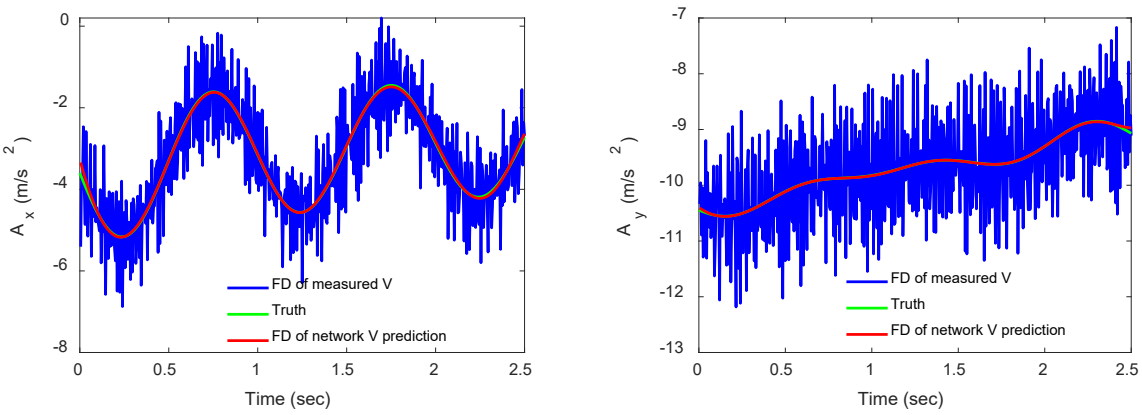
where $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ are weights determining which objective is more important.

The first example of PINNs presented here is a trajectory predicting network that satisfies Newton's second law. Simply put, Equation 19 is expanded and rearranged in a residual form.

$$R_1(t) = 0 = -\frac{\rho A C_D(t)}{2m} V_\infty V_x - \dot{V}_x$$
$$R_2(t) = 0 = -\frac{\rho A C_D(t)}{2m} V_\infty V_y - g - \dot{V}_y \quad (27)$$

After trajectory modeling of Section 3.2 is complete, the sum square of Equation 27, $\sum\left(\boldsymbol{R_1(t)} + \boldsymbol{R_2(t)}\right)^2 \forall\, \boldsymbol{t}$ with unknown $\boldsymbol{C_D(t)}$, is used as a loss function to train another static neural network**.** The new neural net will estimate $\boldsymbol{C_D(t)}$ with trajectory data ($\boldsymbol{V_x, V_y, \dot{V}_x, \dot{V}_y}$) as known, *t* as input, and the sum of Equation 27 as constraints. Acceleration data is needed to evaluate Equation 27. However, trajectory tracking instrumentation such as radar typically provides position and velocity data, and acceleration must be estimated through model fitting. If acceleration and velocity are readily available, the PINN may be trained without first modeling the trajectory as shown in the previous sections.

Since the trajectory fitting network provided only position and velocity, the acceleration is estimated through finite differencing. In Figure 13, finite difference estimates are compared with the trajectory depicted in Figure 12. A small amount of white noise was added to the true velocity data prior to trajectory fitting. The blue lines in Figure 13 show how a small amount of noise is greatly amplified through finite differencing. The green lines eliminate much of the noise by substituting the noisy velocity data into Equation 19 to find acceleration, assuming the drag model is known. The red lines are found by finite differencing the network velocity predictions shown in Figure 12b. The network smooths out the noise and provides acceleration estimates adequate for training the PINN. In Figure 13a, the oscillations in *x* acceleration stem from the harmonic drag coefficient as described at the beginning of Section 3 and shown in Figure 14b.



a. Horizontal acceleration compared

b. Vertical acceleration compared

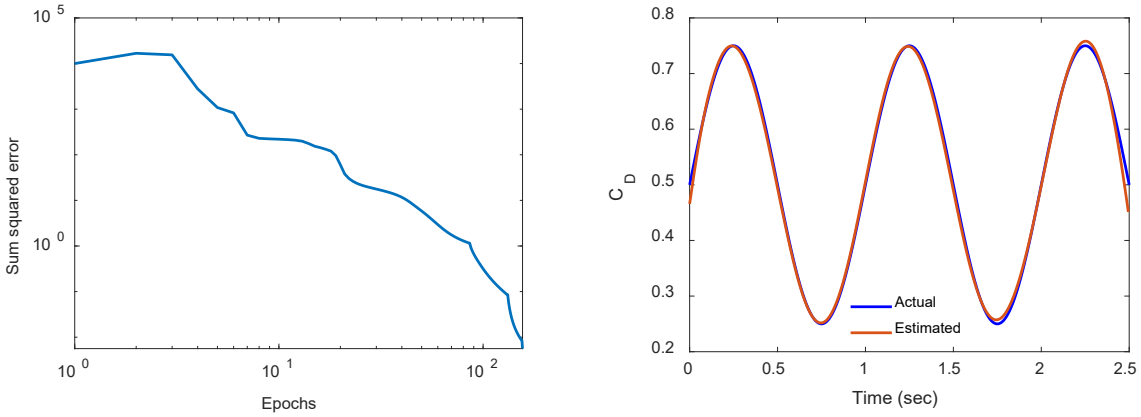
Figure 13. Acceleration network predictions compared to actual after training.

To estimate drag from velocity and acceleration data, a static feedforward network is built with one input, one output, and two hidden layers of seven nodes each. The residual vector was found as the sum of Equation 27. The Jacobian was found by backpropagation with one significant modification. The output unit gradient was found by differentiating the sum of Equation 27, yielding

$$\boldsymbol{\delta_0} = -\frac{\partial C_D^*}{\partial C_D} V_\infty V_x - \frac{\partial C_D^*}{\partial C_D} V_\infty V_z$$

where $\frac{\partial C_D^*}{\partial C_D} = \rho A / 2m$.

Figure 14a shows the convergence over 156 training epochs. Training reduces the error to 0.0056, then stagnates due to noise in the data. Figure 14b compares the network prediction of $C_D(t)$ with the truth model $C_D(t) = \frac{\sin(2\pi t)}{4} + 1/2$. Clearly, the prediction is accurate enough for an analyst to infer a very precise harmonic matching the truth model.



a. Convergence during training

b. Network drag prediction compared to actual

Figure 14. Acceleration network predictions compared to actual after training.

4. HJB Solution by PINN

The HJB equation is one of the established indirect methods to solve optimal control problems. For brevity, Furfaro et al.'s introduction to HJB is summarized.[5] The solution to one of their toy examples is explored using the methods described previously to avoid dependence upon automatic differentiation.

Consider the typical linear quadratic infinite horizon cost function.

$$\min_{\mathbf{u}\in U} J(\mathbf{x},\mathbf{u}) = \int_0^\infty (\mathbf{Q}(\mathbf{x}) + \mathbf{u}^\mathbf{T}\mathbf{R}\mathbf{u})dt \tag{28}$$

subject to

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}) + \mathbf{g}(\mathbf{u})$$

$$\mathbf{x}(0) = \mathbf{x}_0 \tag{29}$$

with $\boldsymbol{x} \in \mathbb{R}^n$ and $\boldsymbol{u} \in \mathbb{R}^m$. The system dynamics $\mathbf{f}(\mathbf{x})$ and $\mathbf{g}(\mathbf{x})$ are assumed to be known. $\mathbf{Q}(\mathbf{x})$ is positive definite, and $\mathbf{R}(\mathbf{x})$ is a symmetric positive definite matrix.

Consider the scalar value function V defined as

$$V(\mathbf{x}) = \inf_{\mathbf{u}\in U} J(\mathbf{x},\mathbf{u})\ . \tag{30}$$

The value function is the unique solution to the HJB equation

$$\sup_{\mathbf{u}\in U}\{-V_x^T\dot{\mathbf{x}} - \mathcal{L}(t,\mathbf{x},\mathbf{u})\} = 0$$

subject to

$$V(\mathbf{0}) = 0\ . \tag{31}$$

The optimal control $\mathbf{u}^*$ is derived with respect to the value function V. That is,

$$\mathbf{u}^* = \underset{\mathbf{u}\in U}{\operatorname{argsup}}\{-V_x^T\dot{\mathbf{x}} - \mathcal{L}(t,\mathbf{x},\mathbf{u})\} = -\frac{1}{2}\mathbf{R}^{-1}\mathbf{g}(\mathbf{x})V_x\ . \tag{32}$$

Once $\mathbf{u}^*$ is derived, its closed analytical form is substituted into Equation 29. Then Equation 29 reduces to a nonlinear PDE that is solved for V.

$$\mathcal{R}(\mathbf{x},V,V_x) = \mathbf{Q}(\mathbf{x}) + V_x^T\mathbf{f}(\mathbf{x}) - \frac{1}{4}V_x^T\mathbf{g}(\mathbf{x})\mathbf{R}^{-1}\mathbf{g}^T(\mathbf{x})V_x = 0$$

subject to

$$V(\mathbf{0}) = 0 \tag{33}$$

From Equation 32, the optimal control is a function of the partial derivative of V with respect to x. I will solve Equation 33 for V using a static feedforward network with n inputs and one output. The network must provide the output signal V as well as derivatives V_x with respect to the network inputs.

Now consider the following toy problem used to demonstrate our HJB solution approach. The starting infinite horizon cost function and system dynamics are as shown in Equations 34 and 35.

$$\min_{\mathbf{u}\in U} J = \int_0^\infty (\mathbf{x}^T\mathbf{x} + u^2)dt \tag{34}$$

subject to

$$\begin{aligned} \dot{x}_1 &= -\mathrm{x}_1 + \mathrm{x}_2 \\ \dot{x}_2 &= -\frac{1}{2}\mathrm{x}_1 - \frac{1}{2}\mathrm{x}_2 + \frac{1}{2}x_1^2x_2 + x_1u \end{aligned} \tag{35}$$

with $x_1 \in [-1,1]$, $x_2 \in [-1,1]$, and $\boldsymbol{u} \in \mathbb{R}$. From Equation 32, the optimal control is $u^* = -x_1V_{x_2}/2$. Substituting the appropriate terms from Equations 34 and 35 into Equation 33, the HJB to solve via PINN is

$$x_1^2 + x_2^2 + (-x_1 + x_2)V_{x_1} + \left(-\frac{1}{2}x_1 - \frac{1}{2}x_2 + \frac{1}{2}x_1^2x_2\right)V_{x_2} - \frac{1}{4}x_1^2V_{x_2}^2 = 0 \tag{36}$$

subject to

$$V(\mathbf{0}) = 0$$

The exact solution of Equation 36 is $V_{exact}(\mathbf{x}) = \frac{x_1^2}{2} + x_2^2$.

Figure 15 shows the network as implemented for this task. There are two inputs, one output, and two hidden layers with seven nodes each. Dotted arrows indicate that backpropagation is used to determine the deltas at the input layer; however, remember that it is actually done *layer by layer* as per Equation 12.

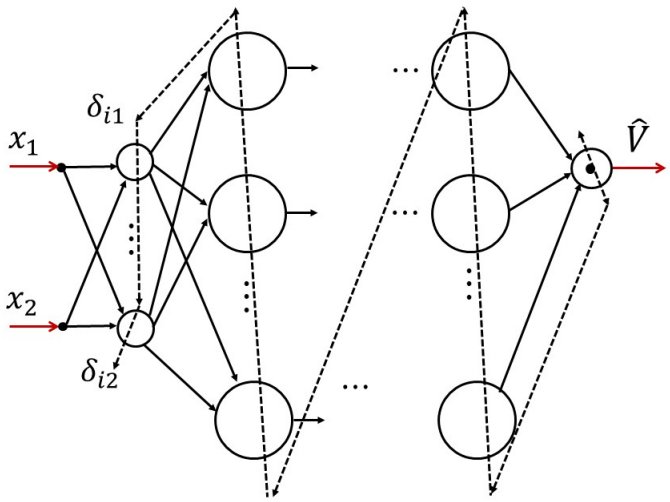


Figure 15. Network for solving the HJB equation.

Once the input layer is reached, Equation 22 is used to find $\hat{V}_{x_1} = \delta_{i1} x_1$ and $\hat{V}_{x_2} = \delta_{i2} x_2$. These quantities will be used in evaluating the network training residual and gradients from Equation 36.

Thus, training proceeds by first discretizing the problem domain $x_1 \in [-1, 1], x_2 \in [-1, 1]$ into 40 evenly spaced points along each dimension for a total grid size of 1600. A final training point is appended at [0, 0] to enforce the boundary condition $V(\mathbf{0}) = 0$. The residual vector is formed by evaluating Equation 36 at the 1600 grid points and multiplying the result by $-2/40$. The boundary condition input is also forward-propagated and its residual is set as $-\hat{V}(\mathbf{0})$.

To determine the Jacobian terms, the sum and chain rules are first applied in differentiating Equation 36

$$\frac{\partial \mathbf{R}}{\partial \mathbf{w}} = \frac{\partial \mathbf{R}}{\partial V_{x_1}} \cdot \frac{\partial V_{x_1}}{\partial \mathbf{w}} + \frac{\partial \mathbf{R}}{\partial V_{x_2}} \cdot \frac{\partial V_{x_2}}{\partial \mathbf{w}} \tag{37}$$

Resulting in

$$\frac{\partial \mathbf{R}}{\partial \mathbf{w}} = (-x_1 + x_2)\frac{\partial V_{x_1}}{\partial \mathbf{w}} + \left(-\frac{1}{2}x_1 - \frac{1}{2}x_2 + \frac{1}{2}x_1^2 x_2 - \frac{1}{2}x_1^2 V_{x_2}\right)\frac{\partial V_{x_2}}{\partial \mathbf{w}} \tag{38}$$

Backpropagation gives us predictions of $\boldsymbol{V_{x_1}}$, $\boldsymbol{V_{x_2}}$, and $\boldsymbol{\partial V/\partial \mathbf{w}}$. The Jacobian requires the terms $\frac{\partial}{\partial \mathbf{w}}\left(\frac{\partial V}{\partial x_1}\right)$ and $\frac{\partial}{\partial \mathbf{w}}\left(\frac{\partial V}{\partial x_2}\right)$. Since backpropagation renders $\boldsymbol{\partial V/\partial \mathbf{w}}$, The order of differentiation is swapped as in Equations 25 and 26. Thus, $\tilde{\mathbf{J}}$ will be formed from $\frac{\partial}{\partial x_1}\left(\frac{\partial V}{\partial w}\right)$, $\frac{\partial}{\partial x_1}\left(\frac{\partial V}{\partial w}\right)$, and $\frac{\partial V(\mathbf{0})}{\partial w}$ for the boundary condition training pair.

This is done by the following:

- Form $\tilde{\mathbf{J}}$ from customary backpropagation terms $\frac{\partial V}{\partial w}$
- Finite difference along each dimension $(\boldsymbol{x_1}, \boldsymbol{x_2})$ to find $\frac{\partial \tilde{\mathbf{J}}}{\partial x_1}$ and $\frac{\partial \tilde{\mathbf{J}}}{\partial x_2}$

  Note: this step presents an extra challenge in that the training pairs are usually collected as

$$\mathbf{q} = \begin{bmatrix} \boldsymbol{x_1(1)} & \cdots & \boldsymbol{x_1(i)} & \cdots \\ \boldsymbol{x_2(1)} & \cdots & \boldsymbol{x_2(j)} & \cdots \end{bmatrix}$$

  such that $(\boldsymbol{x_1(i), x_2(j)})$ spans the set of all possible combinations of grid coordinates. Thus, finite differencing the rows of $\mathbf{q}$ and columns of $\tilde{\mathbf{J}}$ will not render the desired result. This dilemma is handled adroitly in MATLAB by using plaid arrays as follows.*
- Define $\mathbf{x_1}$ and $\mathbf{x_2}$ as the square arrays

$$\mathbf{x_1} = \begin{bmatrix} \boldsymbol{-1} & \boldsymbol{-1} & \boldsymbol{-1} & \cdots \\ \boldsymbol{-1+h} & \boldsymbol{-1+h} & \boldsymbol{-1+h} & \cdots \\ \boldsymbol{-1+2h} & \boldsymbol{-1+2h} & \boldsymbol{-1+2h} & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix}$$

$$\mathbf{x_2} = \begin{bmatrix} \boldsymbol{-1} & \boldsymbol{-1+h} & \boldsymbol{-1+2h} & \cdots \\ \boldsymbol{-1} & \boldsymbol{-1+h} & \boldsymbol{-1+2h} & \cdots \\ \boldsymbol{-1} & \boldsymbol{-1+h} & \boldsymbol{-1+2h} & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix}$$

  where *h* is the step size between grid points.

* Recent unpublished work has shown that this approach can be applied to networks with any number of independent variables by using *nd* plaid arrays and the MATLAB finite difference function along all *n* dimensions.

- Now form $\mathbf{q}$ by stripping columns off $\mathbf{x_1}$ and $\mathbf{x_2}$ and concatenating into two rows

$$\mathbf{q} = \begin{bmatrix} \mathbf{-1} & \mathbf{-1+h} & \mathbf{-1+2h} & \cdots \\ \mathbf{-1} & \mathbf{-1} & \mathbf{-1} & \cdots \end{bmatrix}$$

  This order will be preserved through all calculations. To set up for finite differencing,
- Take column *i* of $\boldsymbol{\partial V/\partial \mathbf{w}}$ and reshape to match the shape of $\mathbf{x_1}$ and $\mathbf{x_2}$ filling a column at a time to form $\hat{\mathbf{J}}_{\boldsymbol{i}}$.
- $\frac{\partial}{\partial \boldsymbol{x_1}}\left(\frac{\partial \boldsymbol{V}}{\partial \boldsymbol{w_i}}\right)$ is the finite difference between rows of $\hat{\mathbf{J}}_{\boldsymbol{i}}$ divided by the finite difference between rows of $\mathbf{x_1}$
- $\frac{\partial}{\partial \boldsymbol{x_2}}\left(\frac{\partial \boldsymbol{V}}{\partial \boldsymbol{w_i}}\right)$ is the finite difference between columns of $\hat{\mathbf{J}}_{\boldsymbol{i}}$ divided by the finite difference between columns of $\mathbf{x_2}$
- Now reshape $\frac{\partial}{\partial \boldsymbol{x_1}}\left(\frac{\partial \boldsymbol{V}}{\partial \boldsymbol{w_i}}\right)$ and $\frac{\partial}{\partial \boldsymbol{x_2}}\left(\frac{\partial \boldsymbol{V}}{\partial \boldsymbol{w_i}}\right)$ into single columns $\tilde{\mathbf{J}}_{\boldsymbol{i}}$ taking a column at a time.
- Column *i* of $\tilde{\mathbf{J}}$ is found by evaluating Equation 38 from inputs and $\tilde{\mathbf{J}}_{\boldsymbol{i}}(\boldsymbol{x_1}) = \frac{\partial \boldsymbol{V}_{\boldsymbol{x_1}}}{\partial \boldsymbol{w_i}}$ and $\tilde{\mathbf{J}}_{\boldsymbol{i}}(\boldsymbol{x_2}) = \frac{\partial \boldsymbol{V}_{\boldsymbol{x_2}}}{\partial \boldsymbol{w_i}}$.

Now, having found $\tilde{\mathbf{J}}$ for the HJB with constraint, training proceeds much like Section 3.2. After a trial weight update, both forward- and backpropagation must be performed to evaluate the new residual from Equation 36.

The HJB network is trained with 1601 training pairs as described above. The sum squared error was reduced below $\mathbf{10^{-6}}$ in 196 epochs**.** Convergence is shown in Figure 16. The network prediction of *V* (in blue) over the training grid is compared to the exact solution (in red) in Figure 17. The prediction is indistinguishable from exact at this axis scaling.

To test the efficacy of our network prediction, the network is ported into a closed-loop simulation such that $\boldsymbol{V_{x_2}}$ is determined by the network from the current system state, and $\boldsymbol{u = -x_1 V_{x_2}/2}$ is used to drive the dynamics given by Equation 35. This closed-loop control was tested for initial conditions at the corners of the problem domain, that is, all combinations of $\boldsymbol{x_1} \in \{\mathbf{-1}, \mathbf{1}\}$ and $\boldsymbol{x_2} \in \{\mathbf{-1}, \mathbf{1}\}$. The closed-loop responses for these four initial conditions are shown in Figure 18 as a) a phase plane plot and b) time responses.

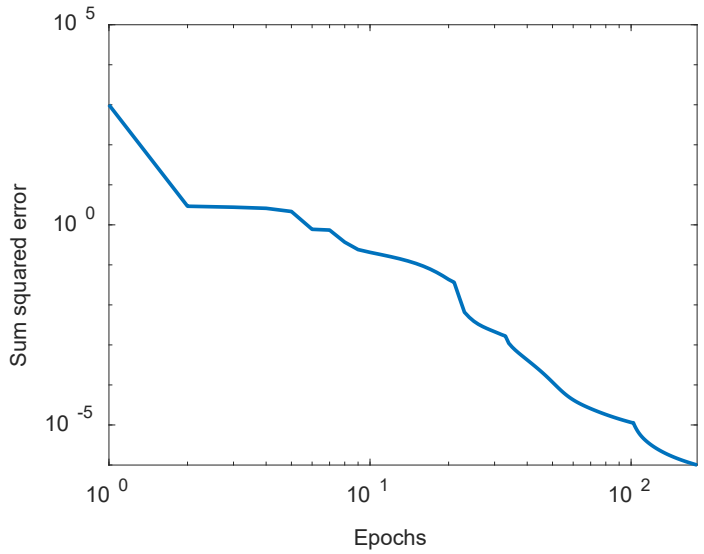


Figure 16. Convergence while training the HJB network using LM.

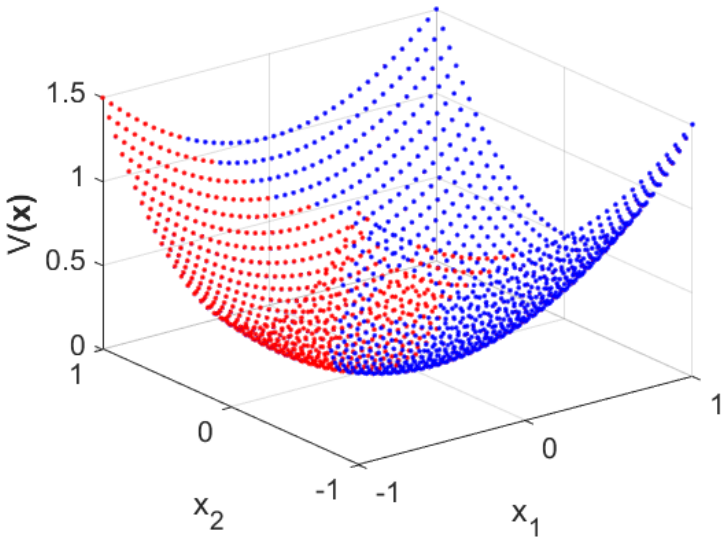
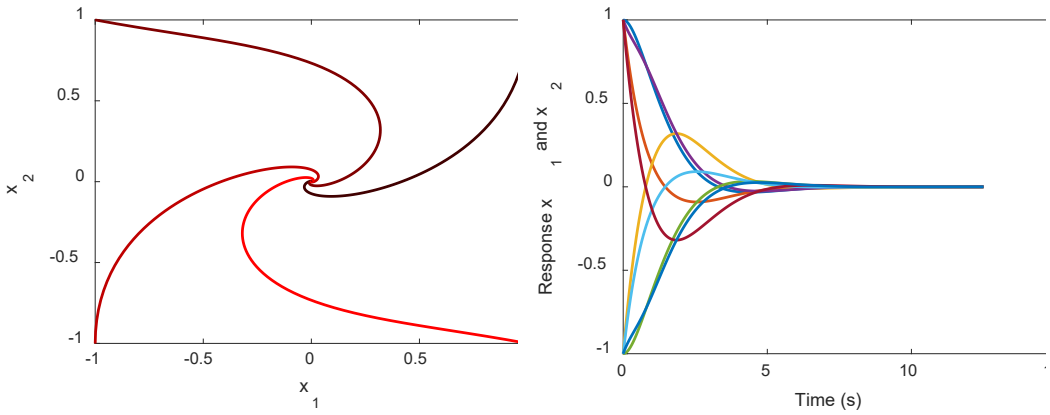


Figure 17. Network predictions of V after training compared to the exact solution of the HJB.



a) Phase plane of closed-loop responses

b) Initial condition response for a variety of initial conditions

Figure 18. Response of the closed-loop HJB controller to initial conditions at the extremes of the problem domain.

All responses display asymptotic stability, returning to the equilibrium point at (0,0). All responses have settling times less than 6 s with half an oscillation of overshoot at most. Thus, the network is able to smoothly predict $\boldsymbol{V}_{\boldsymbol{x_2}}$ and provide the optimal feedback for closed-loop control. Since the training set discretized the operating space, some conditions encountered during the closed-loop test were not part of the training data.

5. Conclusion

This report reviewed feedforward and recurrent neural networks. Several examples were presented using feedforward and dynamic recurrent neural networks to model physical systems. Two low-dimensional examples showed how PINNs use the underlying physics of a system to constrain the solution to a physically meaningful one.

The use of automatic differentiation was avoided and it was shown instead how backpropagation can render the sensitivities needed to train PINNs. In both examples, sensitivities of the prediction with respect to inputs, that is, independent variables, were required to form the modeling residual and its Jacobian in the weight space.

All solutions in this report were rendered with smaller, simpler, shallower networks than those in the references.[5,6] Our networks also trained faster and avoided overtraining. In the future, we hope to solve real-world applications with higher dimensionality using the building blocks presented in this report.

6. References

1. Krose BJA, van der Smagt PP. An introduction to neural networks. J Comput Sci. 1993 Jan 19.
2. Rumelhart DE, Hinton GE, Williams RJ. Learning internal representations by error propagation. Institute for Cognitive Science (ICS), University of California, San Diego; 1985 Sep. ICS Report No.: 8506.
3. Jordan MI, Rumelhart DE. Forward models: supervised learning with a distal teacher. Cogn Sci. 1992;16(3):307–354. https://doi.org/10.1016/0364-0213(92)90036-T
4. Sejnowski TJ. The deep learning revolution. The MIT Press; 2018. ISBN 978-0-262-03803-4.5.
5. Furfaro R, D'Ambrosio A, Schiassi E, Scorsoglio A. Physics-informed neural networks for closed-loop guidance and control in aerospace systems. AIAA 2022-0361. AIAA SCITECH 2022 Forum; 2022 Jan 3–7, San Diego, CA. https://doi.org/10.2514/6.2022-0361
6. Michek N, Mehta P, Huebsch W. Methodology development of a free-flight parameter estimation technique using physics-informed neural networks. 2023 IEEE Aerospace Conference; 2023 Mar 4–11; Big Sky, MT. p. 1–18. doi:10.1109/AERO55745.2023.10115728
7. PyTorch. The Linux Foundation; c2024 [accessed 2024 Dec 6]. https://pytorch.org/
8. Quickstart. The JAX authors; c2024 [accessed 2024 Dec 6]. https://jax.readthedocs.io/en/latest/quickstart.html
9. Hagan MT, Menhaj MB. Training feedforward networks with the Marquardt algorithm. In: IEEE Transactions on Neural Networks. 1994;5(6):989–993. doi:10.1109/72.329697
10. Pearlmutter BA. Gradient calculations for dynamic recurrent neural networks: a survey. In: IEEE Transactions on Neural Networks. 1995;6(5):1212–1228. doi:10.1109/72.410363
11. De Jesus O, Hagan MT. Backpropagation algorithms for a broad class of dynamic networks. In: IEEE Transactions on Neural Networks. 2007;18(1):14–27. doi:10.1109/TNN.2006.882371

12. Franklin GF, Powell JD, Emami-Naeini A. Feedback control of dynamic systems. 3rd ed. Addison Wesley; 1994.

Appendix. Additional Results

3-D Trajectory Reconstruction with Velocity Feedback

The point mass trajectory fitting exercise was repeated with a 3-D point mass flight. This required adjustments to the output dimensions of the neural network, error vector, velocity estimate, and Jacobian matrix. Such an incremental step made us test the flexibility and robustness of our neural net code before applying it to actual data.

Figure A-1 shows the training convergence for this case over 828 epochs. The loss function includes position and velocity errors. Sum squared error after 828 epochs is $9.97(10^{-8})$.

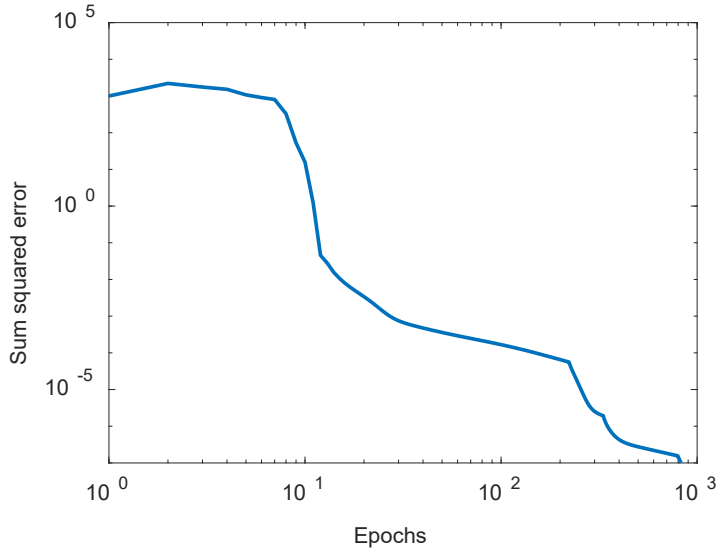


Figure A-1. Training convergence for 3-D trajectory network with velocity predictions.

Figure A-2 shows the actual trajectory and network prediction in 3-D space. Any differences are not discernible at this scaling.

Figure A-3 shows the actual velocity and network prediction after training. The lines are very slightly different where the actual in blue overshadows the network prediction in red.

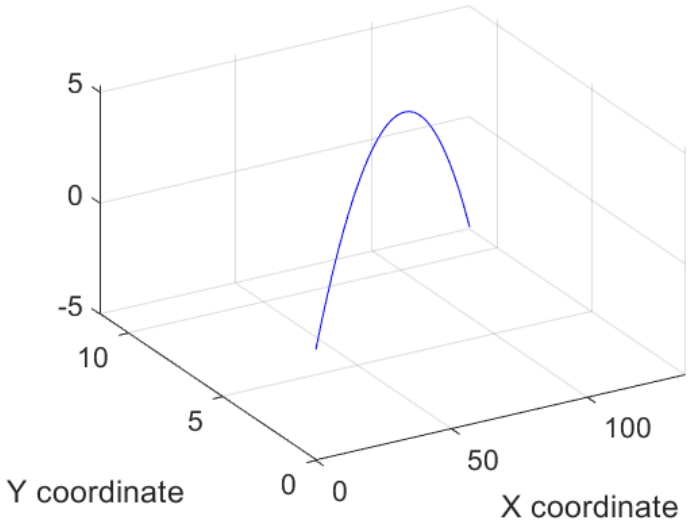


Figure A-2. 3-D trajectory and network prediction after training.

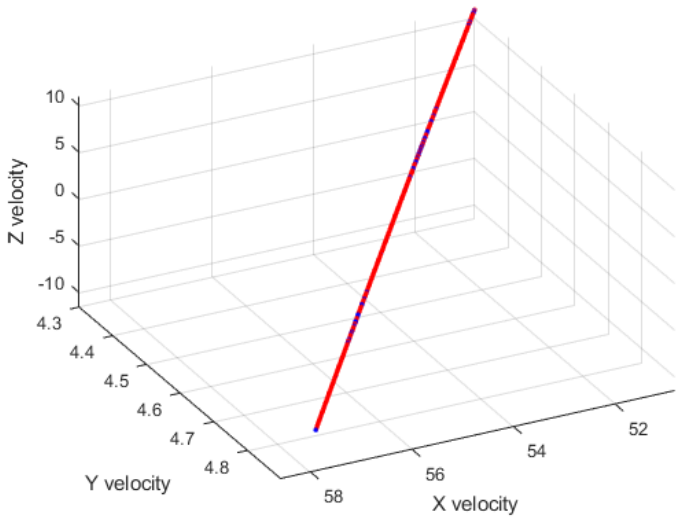


Figure A-3. 3-D trajectory velocity and network velocity estimates after training.

List of Symbols, Abbreviations, and Acronyms

3-D	three-dimensional
HJB	Hamilton–Jacobi–Bellman
LM	Levenberg–Marquardt algorithm
PDE	partial differential equation
PINN	physics-informed neural network

Notional symbols

A	bias
b	bias
δ_j	generalized delta for unit j
$\boldsymbol{E}$	loss function
$f(.)$	activation function
J	training Jacobian matrix
n_l	number of hidden layers
N	number of units in a layer
q	network input vector
R	training residual vector
s_α	$\sin(\alpha)$
w	synaptic weight
x	output of neuron
$\hat{y}$	network output
σ	sum of input signals from synapses

Subscript

h	hidden layer
i	input layer
kj	from unit j to unit k
o	output layer

p	training pair p
$train$	training pairs

Superscript

T	matrix transpose

1 (PDF)	DEFENSE TECHNICAL INFORMATION CTR DTIC OCA
1 (PDF)	DEVCOM ARL FCDD RLB CI TECH LIB