Title

Adaptive Deep Learning for High-Dimensional Hamilton--Jacobi--Bellman Equations

Journal

SIAM Journal on Scientific Computing, 43(2)

ISSN

1064-8275

Authors

Nakamura-Zimmerer, Tenavi
Gong, Qi
Kang, Wei

Publication Date

2021

DOI

10.1137/19m1288802

Peer reviewed

ADAPTIVE DEEP LEARNING FOR HIGH-DIMENSIONAL HAMILTON-JACOBI-BELLMAN EQUATIONS*

TENAVI NAKAMURA-ZIMMERER†, QI GONG†, AND WEI KANG‡

Abstract. Computing optimal feedback controls for nonlinear systems generally requires solving Hamilton-Jacobi-Bellman (HJB) equations, which are notoriously difficult when the state dimension is large. Existing strategies for high-dimensional problems often rely on specific, restrictive problem structures, or are valid only locally around some nominal trajectory. In this paper, we propose a data-driven method to approximate semi-global solutions to HJB equations for general high-dimensional nonlinear systems and compute candidate optimal feedback controls in real-time. To accomplish this, we model solutions to HJB equations with neural networks (NNs) trained on data generated without discretizing the state space. Training is made more effective and data-efficient by leveraging the known physics of the problem and using the partially-trained NN to aid in adaptive data generation. We demonstrate the effectiveness of our method by learning solutions to HJB equations corresponding to the attitude control of a six-dimensional nonlinear rigid body, and nonlinear systems of dimension up to 30 arising from the stabilization of a Burgers'-type partial differential equation. The trained NNs are then used for real-time feedback control of these systems.

Key words. Hamilton-Jacobi-Bellman Equations, Optimal Feedback Control, Nonlinear Dynamical Systems, Deep Learning, Neural Networks, Optimization

AMS subject classifications. 49K15, 49L20, 49N35, 68T05, 90C30, 93C15, 93C20

1. Introduction.

For the optimal control of nonlinear dynamical systems, it is well-known that open-loop controls are not robust to model uncertainty or disturbances. For slowly evolving processes, it is possible to use model predictive control by recomputing the open-loop optimal solutions for relatively short time horizons in the future. However, for most applications one typically desires a feedback control law, as feedback controls are inherently more robust to disturbances. In principle, optimal feedback controllers can synthesized by solving a (discretized) Hamilton-Jacobi-Bellman (HJB) equation, a partial differential equation (PDE) in n spatial dimensions plus time. The size of the discretized problem increases exponentially with n, making direct solution intractable for even moderately large problems. This is the so-called "curse of dimensionality."

For this reason, there is an extensive literature on methods of finding approximate solutions for HJB equations. Some key examples include series expansions [3, 28, 23], level set methods [32], patchy dynamic programming [9, 31], semi-Lagrangian methods [5, 17], method of characteristics and Hopf formula-based algorithms [15, 11, 38], and polynomial approximation [22]. These existing methods suffer one or more of the following drawbacks: the problem's dimension is limited; the accuracy of the solution is hard to verify for general systems; the solution may be valid only in a small region; or the system model must have certain special algebraic structure.

In [24, 25], semi-global solutions to HJB equations are computed by combining the method of characteristics with sparse state space discretization. In this approach,

*Submitted to the journal's Methods and Algorithms for Scientific Computing section September 20, 2019. Preliminary results of this work appeared in the American Control Conference 2020 [30].

Funding: The work of the first and second authors was partially supported with funding from the Defense Advanced Research Projects Agency (DARPA) grant FA8650-18-1-7842.

†Department of Applied Mathematics, Baskin School of Engineering, University of California, Santa Cruz (tenakamu@ucsc.edu, qgong@ucsc.edu).

‡Department of Applied Mathematics, Naval Postgraduate School, Monterery, CA, (wkang@nps.edu).

a two-point boundary value problem (BVP) is solved at each point in a sparse grid. These BVPs can be solved independently, making the algorithm *causality-free*. This property is attractive because the computation does not depend on a grid, and hence they can be applied to high-dimensional problems. The Hopf formula methods [15, 11, 38] also have this property, though it is achieved in a different way and under certain convexity/concavity assumptions. Causality-free methods are usually too slow for online computation, but they are perfectly parallelizable so can be used to generate large data sets offline. Such data sets can then be used to construct faster solutions such as sparse grid interpolants [24, 25] or, as in this paper, neural networks.

Using neural networks (NNs) as a basis for solving HJB equations is not by itself a new idea, and deep learning approaches have led to promising results; see for instance [2, 10, 35, 21, 34, 20]. To the best of our knowledge, state-of-the-art NN-based techniques generally rely on either minimizing the residual of the PDE and (artificial) boundary conditions at randomly sampled collocation points [2, 10, 35, 34]; or, due to computational limitations, approximating the control and/or HJB solution and its gradient in a small neighborhood of a nominal trajectory [21, 20]. In [13, 14], a specialized NN architecture is proposed to solve some classes of Hamilton-Jacobi equations, but this method has yet to be generalized to state-dependent HJB equations arising in optimal control. Deep learning techniques have also been proposed for solving high-dimensional stochastic optimal control problems (see e.g. [18, 19, 4]).

In this paper, we develop a computational method for solving high-dimensional HJB equations and synthesizing candidate optimal feedback controllers. Our approach is data-driven and consists of three main steps. First, we generate a small set of open-loop optimal control solutions using a causality-free algorithm based on Pontryagin's Minimum Principle (PMP). In the second step, we use the data set to train a NN to approximate the solution to the HJB equation, called the *value function*. During training, we supply information about the value function gradient, which encourages the NN to learn the shape of the value function rather than just fitting point data. We also estimate the number of samples needed to obtain a good model. Additional samples are chosen in regions where the value function is difficult to learn, and are obtained quickly with the aid of the NN. In this sense our method involves *adaptive sampling*. Lastly, the accuracy of the NN is verified on independent data generated using the same causality free algorithm from the first step. Unlike other NN-based methods for deterministic HJB equations, our approach does not require computing expensive PDE residuals and the solution is valid over large spatial domains.

As an illustrative example, the method is applied to design an attitude controller of a rigid-body satellite equipped with momentum wheels. This is a highly nonlinear problem with $n = 6$ spatial dimensions and $m = 3$ control inputs. With the proposed method, we obtain a model of the value function with accuracy comparable to that obtained in [25], but require far fewer sample trajectories to do so. Scalability of the method is tested on problems of dimension $n = 10$, 20, and 30 arising from pseudospectral discretization of a Burgers'-type PDE. We show that the method is capable of handling these high-dimensional problems without simplifying the dynamics.

Through these examples, we demonstrate several advantages and potential capabilities of the proposed framework. These include solving HJB equations over semi-global domains with empirically validated levels of accuracy, progressive generation of rich data sets, and computationally efficient nonlinear feedback control for real-time applications. Solution of high-dimensional problems is enabled by efficient and adaptive causality-free data generation, physics-informed learning, and the inherent capacity of NNs for dealing with high-dimensional data.

1.1. Abbreviations and notation. Here we present a brief list of some of the abbreviations, terminology, and notation used in this paper.

OCP	...	optimal control problem
HJB	...	Hamilton-Jacobi-Bellman equation
PMP	...	Pontryagin's Minimum Principle
NN	...	neural network
RMAE	...	relative mean absolute error
RML^2	...	relative mean L^2 error
$\mathcal{D}$	...	data set
μ	...	gradient regularization weight
C	...	adaptive sampling convergence parameter

2. A causality-free method for HJB equations. We consider fixed final time optimal control problems (OCP) of the form

$$\begin{cases} \underset{\boldsymbol{u}(\cdot)\in\mathbb{U}}{\text{minimize}} & J\left[\boldsymbol{u}(\cdot)\right] = F(\boldsymbol{x}(t_f)) + \displaystyle\int_0^{t_f} \mathcal{L}(t,\boldsymbol{x},\boldsymbol{u})dt, \\ \text{subject to} & \dot{\boldsymbol{x}}(t) = \boldsymbol{f}(t,\boldsymbol{x},\boldsymbol{u}), \\ & \boldsymbol{x}(0) = \boldsymbol{x}_0. \end{cases} \tag{2.1}$$

Here $\boldsymbol{x}(t) : [0,t_f] \to \mathbb{X} \subseteq \mathbb{R}^n$ is the state, $\boldsymbol{u}(t,\boldsymbol{x}) : [0,t_f]\times\mathbb{X} \to \mathbb{U} \subseteq \mathbb{R}^m$ is the control, and $\boldsymbol{f}(t,\boldsymbol{x},\boldsymbol{u}) : [0,t_f]\times\mathbb{X}\times\mathbb{U} \to \mathbb{R}^n$ is a Lipschitz continuous vector field. $J\left[\boldsymbol{u}(\cdot)\right]$ is the cost functional which is composed of $F(\boldsymbol{x}(t_f)) : \mathbb{X} \to \mathbb{R}$, the terminal cost, and $\mathcal{L}(t,\boldsymbol{x},\boldsymbol{u}) : [0,t_f]\times\mathbb{X}\times\mathbb{U} \to \mathbb{R}$, the running cost. We assume that the cost functional is convex in $\boldsymbol{x}$ and $\boldsymbol{u}$. In this paper we consider the case where the final time $t_f < \infty$ is fixed.

For a given initial condition $\boldsymbol{x}(0) = \boldsymbol{x}_0$, many numerical methods exist to compute the optimal open-loop solution,

$$\boldsymbol{u} = \boldsymbol{u}^*(t;\boldsymbol{x}_0). \tag{2.2}$$

The open-loop control (2.2) which solves (2.1) is valid for all $t \in [0,t_f]$, but only for the fixed initial condition $\boldsymbol{x}(0) = \boldsymbol{x}_0$. Due to various sources of disturbance and real-time application requirements, for practical implementation one typically desires an optimal control in closed-loop feedback form,

$$\boldsymbol{u} = \boldsymbol{u}^*(t,\boldsymbol{x}), \tag{2.3}$$

which can be evaluated online given any $t \in [0,t_f]$ and a measurement of $\boldsymbol{x} \in \mathbb{X}$.

To compute the optimal feedback control, we follow the standard procedure in dynamic programming (see e.g. [27]) and define the value function $V(t,\boldsymbol{x}) : [0,t_f] \times \mathbb{X} \to \mathbb{R}$ as the optimal cost-to-go of (2.1) starting at $(t,\boldsymbol{x})$. That is,

$$V(t,\boldsymbol{x}) \coloneqq J\left[\boldsymbol{u}^*(\cdot)\right] = \begin{cases} \underset{\boldsymbol{u}(\cdot)\in\mathbb{U}}{\inf} & F(\boldsymbol{y}(t_f)) + \displaystyle\int_t^{t_f} \mathcal{L}(\tau,\boldsymbol{y},\boldsymbol{u})d\tau, \\ \text{s.t.} & \dot{\boldsymbol{y}}(\tau) = \boldsymbol{f}(\tau,\boldsymbol{y},\boldsymbol{u}), \\ & \boldsymbol{y}(t) = \boldsymbol{x}. \end{cases} \tag{2.4}$$

It can be shown that the value function is the unique viscosity solution [12] of the Hamilton-Jacobi-Bellman (HJB) PDE,

$$\begin{cases} -V_t(t,\boldsymbol{x}) - \min\limits_{\boldsymbol{u}\in\mathbb{U}} \left\{\mathcal{L}(t,\boldsymbol{x},\boldsymbol{u}) + [V_{\boldsymbol{x}}(t,\boldsymbol{x})]^T \boldsymbol{f}(t,\boldsymbol{x},\boldsymbol{u})\right\} = 0, \\ V(t_f,\boldsymbol{x}) = F(\boldsymbol{x}), \end{cases} \tag{2.5}$$

where we denote $V_t \coloneqq \partial V/\partial t$ and $V_{\boldsymbol{x}} \coloneqq [\partial V/\partial \boldsymbol{x}]^T$. Note that if the value function is C^2, then it is the unique classical solution of (2.5).

To compute the control given the value function $V(\cdot)$, we start by defining the Hamiltonian

$$\mathcal{H}(t, \boldsymbol{x}, \boldsymbol{\lambda}, \boldsymbol{u}) \coloneqq \mathcal{L}(t, \boldsymbol{x}, \boldsymbol{u}) + \boldsymbol{\lambda}^T \boldsymbol{f}(t, \boldsymbol{x}, \boldsymbol{u}), \tag{2.6}$$

where $\boldsymbol{\lambda}(t) : [0, t_f] \to \mathbb{R}^n$ is the costate. The optimal control satisfies the Hamiltonian minimization condition,

$$\boldsymbol{u}^*(t) = \boldsymbol{u}^*(t, \boldsymbol{x}; \boldsymbol{\lambda}) = \arg\min_{\boldsymbol{u} \in \mathbb{U}} \mathcal{H}(t, \boldsymbol{x}, \boldsymbol{\lambda}, \boldsymbol{u}). \tag{2.7}$$

If we denote the minimized Hamiltonian by $\mathcal{H}^*(t, \boldsymbol{x}, \boldsymbol{\lambda}) \coloneqq \mathcal{H}\left(t, \boldsymbol{x}, \boldsymbol{\lambda}, \boldsymbol{u}^*(t, \boldsymbol{x}; \boldsymbol{\lambda})\right)$, then (2.5) can be expressed as

$$\begin{cases} -V_t(t, \boldsymbol{x}) - \mathcal{H}^*\left(t, \boldsymbol{x}, V_{\boldsymbol{x}}\right) = 0, \\ V(t_f, \boldsymbol{x}) = F(\boldsymbol{x}). \end{cases} \tag{2.8}$$

If (2.8) can be solved (in the viscosity sense), then it provides both necessary and sufficient conditions for optimality. Moreover, the optimal feedback control is computed by substituting

$$\boldsymbol{\lambda}(t) = V_{\boldsymbol{x}}(t, \boldsymbol{x}) \tag{2.9}$$

into (2.7) to get

$$\boldsymbol{u}^*(t, \boldsymbol{x}) = \boldsymbol{u}^*(t, \boldsymbol{x}; V_{\boldsymbol{x}}) = \arg\min_{\boldsymbol{u} \in \mathbb{U}} \mathcal{H}\left(t, \boldsymbol{x}, V_{\boldsymbol{x}}, \boldsymbol{u}\right). \tag{2.10}$$

This means that with $V_{\boldsymbol{x}}(\cdot)$ available, the feedback control is obtained as the solution of an (ideally straightforward) optimization problem.

2.1. Pontryagin's Minimum Principle. To make use of (2.10), we need an efficient way to approximate the value function and its gradient. Like [24, 25], rather than solve the full HJB equation (2.8) on a grid, we exploit the fact that the characteristics of solutions to (2.8) evolve according to a two-point BVP, well-known in optimal control as Pontryagin's Minimum Principle (PMP):

$$\begin{cases} \dot{\boldsymbol{x}}(t) = \mathcal{H}_{\boldsymbol{\lambda}} = \boldsymbol{f}(t, \boldsymbol{x}, \boldsymbol{u}^*(t, \boldsymbol{x}; \boldsymbol{\lambda})), & \boldsymbol{x}(0) = \boldsymbol{x}_0, \\ \dot{\boldsymbol{\lambda}}(t) = -\mathcal{H}_{\boldsymbol{x}}(t, \boldsymbol{x}, \boldsymbol{\lambda}, \boldsymbol{u}^*(t, \boldsymbol{x}; \boldsymbol{\lambda})), & \boldsymbol{\lambda}(t_f) = F_{\boldsymbol{x}}(\boldsymbol{x}(t_f)), \\ \dot{v}(t) = -\mathcal{L}(t, \boldsymbol{x}, \boldsymbol{u}^*(t, \boldsymbol{x}; \boldsymbol{\lambda})), & v(t_f) = F(\boldsymbol{x}(t_f)). \end{cases} \tag{2.11}$$

The two-point BVP provides a necessary condition for optimality. If we further assume that the solution is optimal, then along the characteristic $\boldsymbol{x}(t; \boldsymbol{x}_0)$ we have that

$$\boldsymbol{u}^*(t, \boldsymbol{x}) = \boldsymbol{u}^*(t; \boldsymbol{x}_0), \qquad V(t, \boldsymbol{x}) = v(t; \boldsymbol{x}_0), \qquad V_{\boldsymbol{x}}(t, \boldsymbol{x}) = \boldsymbol{\lambda}(t; \boldsymbol{x}_0). \tag{2.12}$$

In [24, 25], the two-point BVP (2.11) is solved for each point in a sparse grid. Applying (2.12), the value function and its gradient are then calculated using high-dimensional interpolation. This technique is called the sparse grid characteristics method. But even in a sparse grid the number of points grows like $O\left(N(\log N)^{n-1}\right)$, where n is the state dimension and N is the number of grid points in each dimension. Thus one may have to solve a prohibitively large number of BVPs for higher-dimensional problems. Instead of sparse grid interpolation, we use data from solved BVPs to train a NN to approximate the value function. This approach is completely grid-free and hence applicable in high dimensions.

Remark 2.1. In general, the BVP admits multiple solutions which can sometimes be sub-optimal. The characteristics of the value function satisfy (2.11), but there may be other solutions to these equations which are sub-optimal and therefore not characteristics of the value function. In many problems it is also possible for the characteristics to intersect, giving rise to non-smooth value functions and difficulties in applying (2.9).

Optimality of solutions to the BVP can be guaranteed under some convexity conditions (see e.g. [29]). For most dynamical systems it is difficult to verify such conditions globally, but we can guarantee optimality locally around an equilibrium point [28]. Addressing the challenge of global optimality in a broader context is beyond the scope of the present work, so in this paper we assume that solutions to the two-point BVP (2.11) are optimal. Under this assumption, the relationship between PMP and the value function as given in (2.12) holds everywhere.

Note the proposed method can still be applied to problems where this assumption cannot be verified. In such cases PMP remains the prevailing tool for computing candidate optimal solutions, and from these the proposed method will yield a feedback controller which satisfies necessary conditions for optimality.

2.2. Causality-free data generation. While solving the BVP is easier than solving the full HJB equation, we know of no general algorithm that is reliable and fast enough for real-time applications. However, in our approach the real-time feedback control computation is done by a NN which is trained *offline*. Thus we can solve the BVP offline to generate data for training and evaluating such a NN. For this purpose, numerically solving the BVP can be manageable although it may require some parameter tuning. In this paper, we use an implementation of the BVP solver introduced in [26]. This algorithm is based on a three-stage Lobatto IIIa discretization, a collocation formula which provides a solution that is fourth-order accurate. But the algorithm is highly sensitive to the initial guess for $\boldsymbol{x}(t)$ and $\boldsymbol{\lambda}(t)$: there is no guarantee of convergence with an arbitrary initial guess, and in most cases a good initial guess for $\boldsymbol{\lambda}(t)$ cannot be derived from the problem physics. Furthermore, convergence is increasingly dependent on good initializations as we increase the length of the time interval.

To overcome this difficulty, we employ the *time-marching* trick from [24, 25]. This is a continuation technique in which we sequentially extend the solution from an initially short time interval to the final time t_f. Specifically, we choose a sequence of intermediate times

$$0 < t_1 < t_2 < \cdots < t_K = t_f,$$

in which t_1 is small. For the short time interval $[0, t_1]$, the BVP solver converges given most initial guesses near the initial state $\boldsymbol{x}_0$. Then the resulting trajectory is rescaled over the longer time interval $[0, t_2]$. The rescaled trajectory is used as the initial guess to solve the BVP over $0 \le t \le t_2$. We repeat this process until $t_K = t_f$, at which we obtain the full solution. By appropriately tuning the time sequence $\{t_k\}_{k=1}^K$, we can largely overcome the problem of sensitivity to initial guesses.

Computing many such solutions becomes expensive, which means that generating the large data sets necessary to train a NN can be difficult. With this in mind, we use the time-marching trick only to generate a small initial data set, and adaptively adding more points during training. The key to doing this efficiently is simulating the system dynamics using the partially-trained NN to close the loop. The closed-loop trajectory and predicted costate provide good guesses for the optimal state and costate, so that we can immediately solve (2.11) for all of $[0, t_f]$. Besides being more computationally

efficient than time-marching, this approach also requires no parameter tuning. Details are presented in section 4, and numerical comparisons between this method and time-marching are given in subsections 5.2 and 6.2.

As an alternative to either of these two approaches, one could use backpropagation as suggested in [20]. However, this method does not allow one to choose initial conditions independently and so cannot be considered fully causality-free.

3. Neural network approximation of the value function. Neural networks have become a popular tool for modeling high-dimensional functions, since they are not dependent on discretizing the state space. In this paper, we apply NNs to approximate solutions of the HJB equation and evaluate the resulting feedback control in real-time. Specifically, we carry out the following steps:

1. *Initial data generation:* We compute the value function, $V(t, \boldsymbol{x})$, along trajectories $\boldsymbol{x}(t)$ from initial conditions chosen by Monte Carlo sampling. Data is generated by solving the BVP as discussed in subsection 2.2. In this initial data generation step, we require relatively few data points since more data can be added later at little computational cost.
2. *Model training:* Given this data set, we train a NN to approximate the value function. Learning is guided by the underlying structure of the problem, specifically by asking the NN to satisfy Eq. (2.9). In doing so, we regularize the model and make efficient use out of small data sets.
3. *Adaptive data generation:* In the initial training phase we only have a small data set, so the NN only roughly approximates the value function. We now expand the data set by generating data in regions where the value function is likely to be steep or complicated, and thus difficult to learn. Generating additional data is made efficient by good initial guesses obtained from NN-in-the-loop simulations of the system dynamics.
4. *Model refinement and validation:* We continue training the model and increasing the size of the data set until we satisfy some convergence criteria. Then, we check the generalization accuracy of the trained NN on a new set of validation data computed at Monte Carlo sample points.
5. *Feedback control:* We compute the feedback control online by evaluating the gradient of the trained NN and applying PMP. Notably, evaluation of the gradient is *exact* and it is extremely cheap even for large n, enabling real-time implementation in high-dimensional systems.

The crux of the proposed method depends on modeling the value function (2.4) over a semi-global domain $\mathbb{X} \subset \mathbb{R}^n$. We present details of this process in the following subsections. In subsection 3.1, we review the basic structure of feedforward NNs and describe how we train a NN to model the value function. Then in subsection 3.2, we propose a simple way to incorporate information about the known solution structure into training. Finally in subsection 3.4, we demonstrate how to use the trained NN for feedback control. The adaptive data generation scheme is treated separately in section 4. The proposed method is illustrated in section 5 by solving a practical optimal attitude control problem for a rigid body satellite, and then applied to solve larger problems in section 6.

3.1. Feedforward neural networks. In this paper we use multilayer feedforward NNs. While many more sophisticated architectures have been developed for other applications, we find this basic architecture to be more than adequate for our purposes. Let $V(\cdot)$ be the function we wish to approximate and $V^{\mathrm{NN}}(\cdot)$ be its NN representation. Feedforward NNs approximate complicated nonlinear functions by a

composition of simpler functions, namely

$$V(t,\boldsymbol{x}) \approx V^{\mathrm{NN}}(t,\boldsymbol{x}) = g_L \circ \boldsymbol{g}_{L-1} \circ \cdots \circ \boldsymbol{g}_\ell \circ \cdots \circ \boldsymbol{g}_1(t,\boldsymbol{x}),$$

where each layer $\boldsymbol{g}_\ell(\cdot)$ is defined as

$$\boldsymbol{g}_\ell(\boldsymbol{y}) = \sigma_\ell(\boldsymbol{W}_\ell \boldsymbol{y} + \boldsymbol{b}_\ell).$$

Here $\boldsymbol{W}_\ell$ and $\boldsymbol{b}_\ell$ are the weight matrices and bias vectors, respectively. $\sigma_\ell(\cdot)$ represents a nonlinear *activation function* applied component-wise to its argument; popular choices include ReLU, tanh, and other similar functions. In this paper, we use tanh for all the hidden layers. The final layer, $g_L(\cdot)$, is typically linear, so $\sigma_L(\cdot)$ is the identity function.

Let $\boldsymbol{\theta}$ denote the collection of the parameters of the NN, i.e.

$$\boldsymbol{\theta} \coloneqq \{\boldsymbol{W}_\ell, \boldsymbol{b}_\ell\}_{\ell=1}^{L}.$$

The NN is trained by optimizing over the parameters $\boldsymbol{\theta}$ to best approximate $V(t,\boldsymbol{x})$ by $V^{\mathrm{NN}}(t,\boldsymbol{x};\boldsymbol{\theta})$. Specifically, by solving the BVP (2.11) from a set of randomly sampled initial conditions, we get a data set

$$\mathcal{D} = \left\{ \left(t^{(i)}, \boldsymbol{x}^{(i)}\right), V^{(i)} \right\}_{i=1}^{N_d},$$

where $\left(t^{(i)}, \boldsymbol{x}^{(i)}\right)$ are the inputs, $V^{(i)} \coloneqq V\left(t^{(i)}, \boldsymbol{x}^{(i)}\right)$ are the outputs to be modeled, and $i = 1, 2, \ldots, N_d$ are the indices of the data points. In the most naïve setting, the NN is then trained by solving the nonlinear regression problem,

$$\underset{\boldsymbol{\theta}}{\text{minimize}} \quad \frac{1}{N_d} \sum_{i=1}^{N_d} \left[V^{(i)} - V^{\mathrm{NN}}\left(t^{(i)}, \boldsymbol{x}^{(i)}; \boldsymbol{\theta}\right) \right]^2. \tag{3.1}$$

3.2. Physics-informed machine learning. Motivated by the development of physics-informed neural networks [33], we expect that we can improve on the rudimentary loss function in (3.1) by incorporating information about the underlying physics. In [33], and in particular in the context of HJB equations in [2, 10, 35, 34], the known underlying PDE and boundary conditions are imposed by minimizing a residual loss over spatio-temporal collocation points. In this approach, no data is gathered: the PDE is solved directly in the least-squares sense. But this residual must be evaluated over a large number of collocation points and can be rather expensive to compute. Thus we propose a simpler approach of modeling the costate $\boldsymbol{\lambda}(\cdot)$ along with the value function itself, taking full advantage of the ability to gather data along the characteristics of the HJB PDE.

Specifically, we know that the costate must satisfy Eq. (2.9), so we train the NN to minimize

$$\left\| \boldsymbol{\lambda}(t;\boldsymbol{x}) - V_{\boldsymbol{x}}^{\mathrm{NN}}(t,\boldsymbol{x};\boldsymbol{\theta}) \right\|^2,$$

where $V_{\boldsymbol{x}}^{\mathrm{NN}}(\cdot)$ is the gradient of the NN model with respect to the state. This quantity is calculated using automatic differentiation. In machine learning, automatic differentiation is usually used to compute gradients with respect to the model parameters, but is just as easy to apply to computing gradients with respect to inputs. This gradient is exact, so no finite difference approximations are needed. In addition, the computational graph is pre-compiled so evaluating the gradient is cheap.

Costate data $\boldsymbol{\lambda}(t)$ is obtained for each trajectory as a natural product of solving the BVP (2.11). Hence we have the augmented data set,

$$\mathcal{D} = \left\{ \left(t^{(i)}, \boldsymbol{x}^{(i)} \right), \left(V^{(i)}, \boldsymbol{\lambda}^{(i)} \right) \right\}_{i=1}^{N_d}, \tag{3.2}$$

where $\boldsymbol{\lambda}^{(i)} \coloneqq \boldsymbol{\lambda}\left(t^{(i)}; \boldsymbol{x}^{(i)}\right)$. We now define the *physics-informed learning problem*,

$$\underset{\boldsymbol{\theta}}{\text{minimize}} \quad \text{loss}\left(\boldsymbol{\theta}; \mathcal{D}\right) \coloneqq \underset{V}{\text{loss}}\left(\boldsymbol{\theta}; \mathcal{D}\right) + \mu \cdot \underset{\boldsymbol{\lambda}}{\text{loss}}\left(\boldsymbol{\theta}; \mathcal{D}\right). \tag{3.3}$$

Here $\mu \geq 0$ is a scalar weight, the loss with respect to data is

$$\underset{V}{\text{loss}}\left(\boldsymbol{\theta}; \mathcal{D}\right) \coloneqq \frac{1}{N_d} \sum_{i=1}^{N_d} \left[V^{(i)} - V^{\text{NN}}\left(t^{(i)}, \boldsymbol{x}^{(i)}; \boldsymbol{\theta}\right) \right]^2, \tag{3.4}$$

and the gradient regularization is defined as

$$\underset{\boldsymbol{\lambda}}{\text{loss}}\left(\boldsymbol{\theta}; \mathcal{D}\right) \coloneqq \frac{1}{N_d} \sum_{i=1}^{N_d} \left\| \boldsymbol{\lambda}^{(i)} - V_{\boldsymbol{x}}^{\text{NN}}\left(t^{(i)}, \boldsymbol{x}^{(i)}; \boldsymbol{\theta}\right) \right\|^2. \tag{3.5}$$

Following standard practice, when computing the loss functions (3.4) and (3.5), the output data is linearly scaled to the range $[-1, 1]$ to improve the scaling of the optimization problem.

A NN trained to minimize (3.3) learns not just to fit the value data, but it is rewarded for doing so in a way that respects the underlying structure of the problem. Gradient regularization takes the known solution structure into account; this makes it preferable to the usual L^1 or L^2 regularization, which are based on the (heuristic) principle that simpler representations of data are likely to generalize better. Furthermore, we recall that the optimal control depends explicitly on $V_{\boldsymbol{x}}(\cdot)$ – see Eqs. (2.10) and (3.8). Accurate approximation of $V_{\boldsymbol{x}}(\cdot)$ is therefore essential for calculating optimal controls. Our method achieves this through automatic differentiation to compute *exact* gradients and by minimization of the gradient loss term (3.5).

3.3. Model validation. In common practice, one randomly partitions the given data set (3.2) into a training set $\mathcal{D}_{\text{train}}$ and validation set $\mathcal{D}_{\text{val}}$. During training, the loss functions (3.4) and (3.5) are calculated with respect to the training data $\mathcal{D}_{\text{train}}$. We then evaluate the performance of the NN against the validation data $\mathcal{D}_{\text{val}}$, which it did not observe during training. Good validation performance indicates that the NN generalizes well, i.e. it did not overfit the training data. We make the validation test more stringent by generating $\mathcal{D}_{\text{train}}$ and $\mathcal{D}_{\text{val}}$ from *independently drawn* initial conditions, so that the two data sets do not share any part of the same trajectories.

We consider the following error metrics for validation. First, the relative mean absolute error (RMAE) of value function prediction, which is defined as

$$\text{RMAE}(\boldsymbol{\theta}; \mathcal{D}_{\text{val}}) \coloneqq \frac{\sum_{i=1}^{N_d} \left| V^{(i)} - V^{\text{NN}}\left(t^{(i)}, \boldsymbol{x}^{(i)}; \boldsymbol{\theta}\right) \right|}{\sum_{i=1}^{N_d} \left| V^{(i)} \right|}. \tag{3.6}$$

We also measure the relative mean L^2 error (RML^2) of gradient prediction, which is defined as

$$\text{RM}L^2(\boldsymbol{\theta}; \mathcal{D}_{\text{val}}) \coloneqq \frac{\sum_{i=1}^{N_d} \left\| \boldsymbol{\lambda}^{(i)} - V_{\boldsymbol{x}}^{\text{NN}}\left(t^{(i)}, \boldsymbol{x}^{(i)}; \boldsymbol{\theta}\right) \right\|_2}{\sum_{i=1}^{N_d} \left\| \boldsymbol{\lambda}^{(i)} \right\|_2}. \tag{3.7}$$

We consider these error metrics instead of pointwise relative errors in order to emphasize predictive accuracy in regions where a lot of control effort is needed. This is important because we are interested in designing nonlinear controllers which are effective and efficient far away from the equilibrium.

3.4. Neural network in the closed-loop system. Once the NN is trained, evaluating $V_{\boldsymbol{x}}^{\mathrm{NN}}(t,\boldsymbol{x})$ at new inputs is highly efficient. Moreover, since we minimized the gradient loss (3.5) during training, we also expect $V_{\boldsymbol{x}}^{\mathrm{NN}}(t,\boldsymbol{x})$ to approximate the true gradient well. At runtime, whenever the feedback control needs to be computed, we evaluate $V_{\boldsymbol{x}}^{\mathrm{NN}}(t,\boldsymbol{x})$ and then solve (2.10) based on this approximation.

For many problems of interest, the optimization problem (2.10) admits an analytic or semi-analytic solution. In particular, for the important class of control affine systems with running cost convex in $\boldsymbol{u}$, we can solve (2.10) analytically. Suppose that the system dynamics can be written in the form

$$\dot{\boldsymbol{x}} = \boldsymbol{f}(t,\boldsymbol{x}) + \boldsymbol{g}(t,\boldsymbol{x})\boldsymbol{u},$$

where $\boldsymbol{f}(t,\boldsymbol{x}) : [0,t_f]\times\mathbb{X}\to\mathbb{R}^n$, $\boldsymbol{g}(t,\boldsymbol{x}) : [0,t_f]\times\mathbb{X}\to\mathbb{R}^{n\times m}$, and the control is unconstrained. Further, suppose that the running cost is of the form

$$\mathcal{L}(t,\boldsymbol{x},\boldsymbol{u}) = h(t,\boldsymbol{x}) + \boldsymbol{u}^T\boldsymbol{W}\boldsymbol{u},$$

for some convex function $h(t,\boldsymbol{x}) : [0,t_f]\times\mathbb{X}\to\mathbb{R}$ and some positive definite weight matrix $\boldsymbol{W}\in\mathbb{R}^{m\times m}$. Then the Hamiltonian is

$$\mathcal{H}(t,\boldsymbol{x},\boldsymbol{\lambda},\boldsymbol{u}) = h(t,\boldsymbol{x}) + \boldsymbol{u}^T\boldsymbol{W}\boldsymbol{u} + \boldsymbol{\lambda}^T\boldsymbol{f}(t,\boldsymbol{x}) + \boldsymbol{\lambda}^T\boldsymbol{g}(t,\boldsymbol{x})\boldsymbol{u}.$$

Now we apply PMP, which for unconstrained control requires

$$\boldsymbol{0}_{m\times 1} = \mathcal{H}_{\boldsymbol{u}}(t,\boldsymbol{x},\boldsymbol{\lambda},\boldsymbol{u}^*) = 2\boldsymbol{W}\boldsymbol{u}^* + \boldsymbol{g}^T(t,\boldsymbol{x})\boldsymbol{\lambda}.$$

Letting $\boldsymbol{\lambda} = V_{\boldsymbol{x}}(t,\boldsymbol{x})$ and solving for $\boldsymbol{u}^*$ yields the optimal feedback control law in explicit form:

$$\boldsymbol{u}^*(t,\boldsymbol{x};V_{\boldsymbol{x}}) = -\frac{1}{2}\boldsymbol{W}^{-1}\boldsymbol{g}^T(t,\boldsymbol{x})V_{\boldsymbol{x}}(t,\boldsymbol{x}). \tag{3.8}$$

The resulting NN controller is then simply

$$\boldsymbol{u}^{\mathrm{NN}}(t,\boldsymbol{x}) = \boldsymbol{u}^*\left(t,\boldsymbol{x};V_{\boldsymbol{x}}^{\mathrm{NN}}\right) = -\frac{1}{2}\boldsymbol{W}^{-1}\boldsymbol{g}^T(t,\boldsymbol{x})V_{\boldsymbol{x}}^{\mathrm{NN}}(t,\boldsymbol{x}). \tag{3.9}$$

4. Adaptive sampling and model refinement. Since generating just a single data point requires solving a challenging BVP, it can be expensive to generate large data sets which adequately represent the value function. This necessitates training using limited data and a method to generate new data in a smart and efficient way. In this paper, effective training with small data sets is accomplished by incorporating information about the costate as discussed in subsection 3.2, but also by combining progressive data generation with an efficient adaptive sampling technique.

Optimization methods in machine learning (see e.g. [6] for a comprehensive survey) are typically divided into second and first order methods. Second order methods like L-BFGS [8] rely on accurate gradient computations, and hence generally have to use the entire data set. For this reason they are often referred to as batch or full-batch methods. On the other hand, first order methods based on stochastic gradient descent

(SGD) use only small subsets, or mini-batches, of the full data set. That is, at each optimization iteration k, the loss functions in (3.1) and (3.3) are evaluated only on a subset $\mathcal{S}_k \subset \mathcal{D}_{\text{train}}$ with $|\mathcal{S}_k| \ll |\mathcal{D}_{\text{train}}|$. Here $|\mathcal{D}|$ denotes the number of data points in a data set $\mathcal{D}$. Although second order methods converge much more quickly than first order methods, the necessary gradient calculations are prohibitively expensive for large data sets. Consequently, SGD variants have become the de facto standard for machine learning applications.

But in the context of deep learning, our NNs are small and data sets smaller. Thus we expect second order methods to be superior for our purposes. With a small initial data set, which we denote by $\mathcal{D}^1_{\text{train}}$, we find that training a low-fidelity model is very fast using L-BFGS. After this initial round, we want to increase the size of the data set so that it better captures the features of the value function. We then continue training the model using this larger data set, $\mathcal{D}^2_{\text{train}}$. We continue this process until some convergence conditions are satisfied.

Our approach is similar to and inspired by a progressive batching method proposed in [7]. The primary difference is that the problem addressed in [7] is a standard machine learning problem, where a massive data set is available from the start. This allows one to increase the sample size every few iterations, and take a completely different sample from the available data. In our problem, start with only a small amount of data and we can generate more as we go, but since data generation is expensive, we would like to generate only as much as is needed.

4.1. Convergence test and sample size selection. In this section we derive a convergence test and sample size selection scheme for the purpose of progressive data generation. To start, suppose that the internal optimizer (e.g. L-BFGS) converges in optimization round r and let $\mathcal{D}^r_{\text{train}}$ be the available training data set. Given convergence of the internal optimizer, the first order necessary condition for optimality holds, so

$$\left\| \frac{\partial \text{loss}}{\partial \boldsymbol{\theta}} (\boldsymbol{\theta}; \mathcal{D}^r_{\text{train}}) \right\| \ll 1. \tag{4.1}$$

Here $\text{loss}(\cdot)$ is the physics-informed loss defined in Eq. (3.3), and $\frac{\partial \text{loss}}{\partial \boldsymbol{\theta}}(\cdot)$ is its gradient with respect to the NN parameters $\boldsymbol{\theta}$. For true first order optimality, we would like the gradient to be small when evaluated over the entire continuous domain of interest, $[0, t_f] \times \mathbb{X}$. In other words, we want

$$\left\| \frac{\partial \text{loss}}{\partial \boldsymbol{\theta}} (\boldsymbol{\theta}; [0, t_f] \times \mathbb{X}) \right\| \ll 1, \tag{4.2}$$

where the Monte Carlo sums in Eqs. (3.4) and (3.5) become integrals in the limit as the size of the data set approaches infinity.

The simplest way to see if (4.2) holds is to generate a validation data set $\mathcal{D}_{\text{val}}$. Then using the fact that $\frac{\partial \text{loss}}{\partial \boldsymbol{\theta}} (\boldsymbol{\theta}; \mathcal{D}_{\text{val}}) \to \frac{\partial \text{loss}}{\partial \boldsymbol{\theta}} (\boldsymbol{\theta}; [0, t_f] \times \mathbb{X})$ in the limit as $|\mathcal{D}_{\text{val}}| \to \infty$, one checks if, for example,

$$\left\| \frac{\partial \text{loss}}{\partial \boldsymbol{\theta}} (\boldsymbol{\theta}; \mathcal{D}_{\text{val}}) \right\| < \epsilon, \tag{4.3}$$

for some small parameter $\epsilon > 0$. Convergence tests like (4.3) are standard in machine learning and are useful for testing generalization performance. But for many practical problems, it may be too expensive to generate enough validation data to make the

test meaningful. More importantly, such tests provides no clear guidance in selecting the sample size $\left|\mathcal{D}_{\text{train}}^{r+1}\right|$ should they not be satisfied.

In this paper, we use validation tests to quantify model accuracy after training is complete (see subsection 3.3). Indeed, the ability to empirically validate solutions is a key benefit of the causality-free approach. For the purpose of determining convergence between training rounds, however, we propose a different statistically motivated test which provides information on choosing $\left|\mathcal{D}_{\text{train}}^{r+1}\right|$. The idea is simple: since we already assume (4.1) holds, then to ensure that (4.2) is also satisfied, it suffices to check that the error in approximating (4.2) by (4.1) is relatively small.

To motivate this more rigorously, consider a finite sample set $\mathcal{D} \subset [0, t_f] \times \mathbb{X}$ with fixed size $|\mathcal{D}|$, and assume that the sample points $\left(t^{(i)}, \boldsymbol{x}^{(i)}\right) \in \mathcal{D}$ are independent and identically distributed (i.i.d.)[1]. By design, if $\left(t^{(i)}, \boldsymbol{x}^{(i)}\right)$ are i.i.d. then the sample gradient $\frac{\partial \text{loss}}{\partial \boldsymbol{\theta}}(\boldsymbol{\theta}; \mathcal{D})$ is an unbiased estimator for the true gradient (evaluated over the entire continuous domain). That is,

$$\mathbb{E}_{\mathcal{D}}\left[\frac{\partial \text{loss}}{\partial \boldsymbol{\theta}}(\boldsymbol{\theta}; \mathcal{D})\right] = \frac{\partial \text{loss}}{\partial \boldsymbol{\theta}}(\boldsymbol{\theta}; [0, t_f] \times \mathbb{X}), \tag{4.4}$$

where $\mathbb{E}_{\mathcal{D}}[\cdot] \coloneqq \mathbb{E}_{\mathcal{D} \subset [0,t_f] \times \mathbb{X}}[\cdot]$ denotes the population mean over all possible finite sample sets $\mathcal{D} \subset [0, t_f] \times \mathbb{X}$ with fixed size $|\mathcal{D}|$. Intuitively, (4.4) implies that if (4.1) holds, then on average we also have (4.2), as desired. But we must control the mean square error (MSE) of the estimator, which is given by

$$\begin{aligned} \text{MSE}\left[\frac{\partial \text{loss}}{\partial \boldsymbol{\theta}}(\boldsymbol{\theta}; \mathcal{D})\right] &\coloneqq \mathbb{E}_{\mathcal{D}}\left[\left\|\frac{\partial \text{loss}}{\partial \boldsymbol{\theta}}(\boldsymbol{\theta}; \mathcal{D}) - \frac{\partial \text{loss}}{\partial \boldsymbol{\theta}}(\boldsymbol{\theta}; [t_0, t_f] \times \mathbb{X})\right\|^2\right] \\ &= \mathbb{E}_{\mathcal{D}}\left[\sum_{j=1}^{|\boldsymbol{\theta}|}\left(\frac{\partial \text{loss}}{\partial \theta_j}(\boldsymbol{\theta}; \mathcal{D}) - \frac{\partial \text{loss}}{\partial \theta_j}(\boldsymbol{\theta}; [t_0, t_f] \times \mathbb{X})\right)^2\right]. \end{aligned} \tag{4.5}$$

To simplify this, using linearity of the expectation we obtain

$$\text{MSE}\left[\frac{\partial \text{loss}}{\partial \boldsymbol{\theta}}(\boldsymbol{\theta}; \mathcal{D})\right] = \sum_{j=1}^{|\boldsymbol{\theta}|} \text{Var}_{\mathcal{D} \subset [0,t_f] \times \mathbb{X}}\left[\frac{\partial \text{loss}}{\partial \theta_j}(\boldsymbol{\theta}; \mathcal{D})\right],$$

and then by construction of the loss function,

$$\text{MSE}\left[\frac{\partial \text{loss}}{\partial \boldsymbol{\theta}}(\boldsymbol{\theta}; \mathcal{D})\right] = \sum_{j=1}^{|\boldsymbol{\theta}|} \text{Var}_{\mathcal{D} \subset [0,t_f] \times \mathbb{X}}\left[\frac{1}{|\mathcal{D}|}\sum_{i=1}^{|\mathcal{D}|}\frac{\partial \text{loss}}{\partial \theta_j}\left(\boldsymbol{\theta}; \left(t^{(i)}, \boldsymbol{x}^{(i)}\right)\right)\right].$$

[1] In practice, while initial conditions are i.i.d., points at future times lie along the optimal trajectories coming from these initial conditions and are thus spatially correlated. Adaptive sampling (see subsection 4.2) also introduces sample dependence. This likely reduces sample variance compared to i.i.d. data, but we still find the numerical tests useful for providing sample size guidelines. In addition, if we learn only the initial-time value function as in section 5, then sample independence can be upheld if we forego adaptive sample placement.

Using the simplifying assumption that $\left(t^{(i)}, \boldsymbol{x}^{(i)}\right)$ are i.i.d., this becomes

$$\begin{aligned}\mathrm{MSE}\left[\frac{\partial \mathrm{loss}}{\partial \boldsymbol{\theta}}(\boldsymbol{\theta};\mathcal{D})\right] &= \frac{1}{|\mathcal{D}|^2}\sum_{j=1}^{|\boldsymbol{\theta}|}\sum_{i=1}^{|\mathcal{D}|}\mathrm{Var}_{(t,\boldsymbol{x})\in[0,t_f]\times\mathbb{X}}\left[\frac{\partial \mathrm{loss}}{\partial \theta_j}(\boldsymbol{\theta};(t,\boldsymbol{x}))\right] \\ &= \frac{1}{|\mathcal{D}|}\sum_{j=1}^{|\boldsymbol{\theta}|}\mathrm{Var}_{(t,\boldsymbol{x})\in[0,t_f]\times\mathbb{X}}\left[\frac{\partial \mathrm{loss}}{\partial \theta_j}(\boldsymbol{\theta};(t,\boldsymbol{x}))\right]. \qquad (4.6)\end{aligned}$$

If the estimation error is small, then the sample mean is likely to be a good approximation of the true mean. Hence we expect that $\left\|\frac{\partial \mathrm{loss}}{\partial \boldsymbol{\theta}}(\boldsymbol{\theta};[0,t_f]\times\mathbb{X})\right\|$ will also be small as desired. To this end, we require that the root MSE not be too large compared to the expected gradient. Specifically, we check if

$$\sqrt{\mathrm{MSE}\left[\frac{\partial \mathrm{loss}}{\partial \boldsymbol{\theta}}(\boldsymbol{\theta};\mathcal{D})\right]} \le C\left\|\mathbb{E}_{\mathcal{D}}\left[\frac{\partial \mathrm{loss}}{\partial \boldsymbol{\theta}}(\boldsymbol{\theta};\mathcal{D})\right]\right\|_1, \qquad (4.7)$$

where $C>0$ is a scalar parameter. On the right hand side we use the L^1 norm instead of the L^2 as it is less sensitive to outliers in the loss gradient. In practice we find that this makes the test less likely to suggest unreasonably large sample sizes.

In practice, evaluating of the true population variances on the left hand side of (4.7) is computationally intractable. But we can approximate these by the corresponding *sample* variances[2] taken over all data $\left(t^{(i)}, \boldsymbol{x}^{(i)}\right) \in \mathcal{D}^r_{\mathrm{train}}$, which we denote by $\mathrm{Var}_{\mathcal{D}^r_{\mathrm{train}}}[\cdot] \coloneqq \mathrm{Var}_{\left(t^{(i)},\boldsymbol{x}^{(i)}\right)\in\mathcal{D}^r_{\mathrm{train}}}[\cdot]$:

$$\mathrm{MSE}\left[\frac{\partial \mathrm{loss}}{\partial \boldsymbol{\theta}}(\boldsymbol{\theta};\mathcal{D})\right] \approx \frac{1}{|\mathcal{D}^r_{\mathrm{train}}|}\sum_{j=1}^{|\boldsymbol{\theta}|}\mathrm{Var}_{\mathcal{D}^r_{\mathrm{train}}}\left[\frac{\partial \mathrm{loss}}{\partial \theta_j}\left(\boldsymbol{\theta};\left(t^{(i)},\boldsymbol{x}^{(i)}\right)\right)\right].$$

Similarly, we approximate the expected gradient on the right hand side of (4.7) by the sample gradient and arrive at the following practical convergence criterion:

$$\sqrt{\sum_{j=1}^{|\boldsymbol{\theta}|}\mathrm{Var}_{\mathcal{D}^r_{\mathrm{train}}}\left[\frac{\partial \mathrm{loss}}{\partial \theta_j}\left(\boldsymbol{\theta};\left(t^{(i)},\boldsymbol{x}^{(i)}\right)\right)\right]} \le C\left\|\frac{\partial \mathrm{loss}}{\partial \boldsymbol{\theta}}\left(\boldsymbol{\theta};\mathcal{D}^r_{\mathrm{train}}\right)\right\|_1\sqrt{|\mathcal{D}^r_{\mathrm{train}}|}. \qquad (4.8)$$

If the convergence test (4.8) is satisfied, then it is likely that the expected gradient $\left\|\frac{\partial \mathrm{loss}}{\partial \boldsymbol{\theta}}(\boldsymbol{\theta};[0,t_f]\times\mathbb{X})\right\|$ is also small. In other words, we expect that the parameters $\boldsymbol{\theta}$ satisfies the first order optimality conditions evaluated over the entire domain, so we can stop optimization. Satisfaction of (4.8) does not imply that the trained model is good – merely that seeing more data would probably not improve it significantly. On the other hand, when the criterion is not met, then it guides us in selecting the next sample size $\left|\mathcal{D}^{r+1}_{\mathrm{train}}\right|$. Concretely, suppose that the ratio of the sample variance to the sample gradient doesn't change significantly by increasing the size of the data set, i.e.

$$\frac{\sqrt{\sum_{j=1}^{|\boldsymbol{\theta}|}\mathrm{Var}_{\mathcal{D}^{r+1}_{\mathrm{train}}}\left[\frac{\partial \mathrm{loss}}{\partial \theta_j}\left(\boldsymbol{\theta};\left(t^{(i)},\boldsymbol{x}^{(i)}\right)\right)\right]}}{\left\|\frac{\partial \mathrm{loss}}{\partial \boldsymbol{\theta}}\left(\boldsymbol{\theta};\mathcal{D}^{r+1}_{\mathrm{train}}\right)\right\|_1} \approx \frac{\sqrt{\sum_{j=1}^{|\boldsymbol{\theta}|}\mathrm{Var}_{\mathcal{D}^{r}_{\mathrm{train}}}\left[\frac{\partial \mathrm{loss}}{\partial \theta_j}\left(\boldsymbol{\theta};\left(t^{(i)},\boldsymbol{x}^{(i)}\right)\right)\right]}}{\left\|\frac{\partial \mathrm{loss}}{\partial \boldsymbol{\theta}}\left(\boldsymbol{\theta};\mathcal{D}^{r}_{\mathrm{train}}\right)\right\|_1}.$$

[2]Computing a large number of individual gradients can still be too costly, so we often evaluate sample variances over a smaller subset of the training data.

Then the appropriate choice of $\left|\mathcal{D}_{\text{train}}^{r+1}\right|$ to satisfy (4.8) after the next round is such that

$$M\left|\mathcal{D}_{\text{train}}^{r}\right| \geq \left|\mathcal{D}_{\text{train}}^{r+1}\right| \geq \frac{\sum_{j=1}^{|\boldsymbol{\theta}|} \operatorname{Var}_{\mathcal{D}_{\text{train}}^{r}}\left[\frac{\partial \text{loss}}{\partial \theta_j}\left(\boldsymbol{\theta};\left(t^{(i)}, \boldsymbol{x}^{(i)}\right)\right)\right]}{\left(C\left\|\frac{\partial \text{loss}}{\partial \boldsymbol{\theta}}\left(\boldsymbol{\theta}; \mathcal{D}_{\text{train}}^{r}\right)\right\|_1\right)^2}, \tag{4.9}$$

where $M > 1$ is a scalar parameter which prevents the data set size from growing too quickly. Throughout this paper we use $M = 2$.

The convergence test (4.8) and sample size selection scheme (4.9) derived above are close to that used in [7], except that we employ the L^1 norm of the sample gradient in the denominator instead of the L^2 norm. We prefer the L^1 norm because it is less sensitive to outliers in the loss gradient. Intuitively, this improves robustness by making the test less likely to suggest unreasonably large sample sizes. We also contribute a different derivation, coming from the perspective of progressive data generation as opposed to sampling from a large pre-existing data set. Finally, like [7] our results are not specific to learning solutions to the HJB equation. They can be applied to many data-driven optimization problems where data is scarce but can be generated over time. Notably, these results facilitate the use of existing algorithms for second order and constrained optimization in such applications.

4.2. Adaptive data generation with NN warm start. The sample size selection criterion (4.9) we propose indicates how many data are necessary to satisfy the convergence test (4.8), assuming a uniform sampling from the domain. In practice, since all the data we generate will be new, we can choose to generate new data where it is needed most, hence the term *adaptive sampling*. This condition for generating new data can be interpreted in many ways. In this paper, we concentrate samples where $\left\|V_{\boldsymbol{x}}^{\text{NN}}(\cdot)\right\|$ is large. Regions of the value function with large gradients tend to be steep or complicated, and thus may benefit from having more data to learn from. Furthermore, these regions correspond to places where the control effort is large and hence we would like controllers to be especially accurate there.

Specifically, for each initial condition we want to integrate, we can first randomly sample a set of N_c candidate initial conditions from $\mathbb{X}$. A quick pass through the NN yields the predicted gradient at all candidate points:

$$\left\{V_{\boldsymbol{x}}^{\text{NN}}\left(0, \boldsymbol{x}_0^{(i)}\right)\right\}_{i=1}^{N_c}.$$

We then choose the point(s) with the largest predicted gradient norms and solve the BVP (2.11) for each of these. To aid in solving these BVPs, instead of using the time-marching trick described in subsection 2.2, we simulate the system dynamics using the partially-trained NN as the closed-loop controller and predicting $\boldsymbol{\lambda}(t) \approx V_{\boldsymbol{x}}^{\text{NN}}(t, \boldsymbol{x}(t))$ along the trajectory. In most cases, this yields an approximate solution which is reasonably close to the optimal state and costate. By supplying this trajectory as an initial guess to the BVP solver, we then quickly and reliably obtain a solution to the BVP for the full time interval $[0, t_f]$. This process is repeated for new initial conditions until we obtain the desired amount of data (each trajectory may contain hundreds of data points). We refer to this technique as a *NN warm start*. A summary of the full training procedure is given in Algorithm 4.1.

Algorithm 4.1 enables us to build up a rich data set and a high-fidelity model of $V(\cdot)$. Moreover, the data set is not constrained to lie within a small neighborhood of some nominal trajectory. It can contain points from the entire domain $\mathbb{X}$, and we

Algorithm 4.1 Adaptive sampling and model refinement

Generate $\mathcal{D}^1_{\text{train}}$ using time-marching
for $r = 1, 2, \ldots$ **do**
  Solve (3.3) for $\boldsymbol{\theta}$
  if (4.8) is satisfied **then**
    return optimized parameters $\boldsymbol{\theta}$ and NN validation accuracy
  else
    while (4.9) is not satisfied **do**
      Sample candidate initial conditions $\boldsymbol{x}_0^{(i)}$, $i = 1, \ldots, N_c$, from $\mathbb{X}$
      In parallel, predict $\left\| V^{\text{NN}}_{\boldsymbol{x}}\left(0, \boldsymbol{x}_0^{(i)}\right)\right\|$, $i = 1, \ldots, N_c$
      Choose the initial condition(s) with largest predicted gradient norm and use NN warm start to solve the BVP (2.11)
      Add the resulting trajectories to $\mathcal{D}^{r+1}_{\text{train}}$
    end while
  end if
end for

can concentrate more data near complicated features of the value function. As we progressively refine the NN model, we can adjust the gradient loss weight μ, as well as other hyperparameters such as the internal optimizer convergence tolerance and the number of terms in the L-BFGS Hessian approximation. As the NN is already partially-trained, fewer iterations should be needed for convergence in each round so we can afford to make each iteration more expensive.

5. Application to rigid body attitude control. To illustrate the capabilities of proposed method, we consider the six-state rigid body model of a satellite studied by Kang and Wilcox [24, 25]. With the sparse grid characteristics method, they interpolate the value function at initial time, $V(t = 0, \boldsymbol{x})$, and use this for moving horizon feedback control of the nonlinear system. We use their successful results as a baseline for evaluating our method.

Let $\{\boldsymbol{e}_1, \boldsymbol{e}_2, \boldsymbol{e}_3\}$ be an inertial frame of orthonormal vectors and let $\{\boldsymbol{e}'_1, \boldsymbol{e}'_2, \boldsymbol{e}'_3\}$ be a body frame. The state of the satellite is then written as $\boldsymbol{x} = \begin{pmatrix}\boldsymbol{v} & \boldsymbol{\omega}\end{pmatrix}$. Here $\boldsymbol{v}$ is the attitude of the satellite represented in Euler angles,

$$\boldsymbol{v} = \begin{pmatrix}\phi & \theta & \psi\end{pmatrix}^T,$$

in which ϕ, θ, and ψ are the angles of rotation around $\boldsymbol{e}'_1$, $\boldsymbol{e}'_2$, and $\boldsymbol{e}'_3$, respectively, in the order $(1, 2, 3)$. These are also commonly called roll, pitch, and yaw. $\boldsymbol{\omega}$ denotes the angular velocity in the body frame,

$$\boldsymbol{\omega} = \begin{pmatrix}\omega_1 & \omega_2 & \omega_3\end{pmatrix}^T.$$

For details see [16]. The state dynamics are

$$\begin{pmatrix}\dot{\boldsymbol{v}} \\ \boldsymbol{J}\dot{\boldsymbol{\omega}}\end{pmatrix} = \begin{pmatrix}\boldsymbol{E}(\boldsymbol{v})\boldsymbol{\omega} \\ \boldsymbol{S}(\boldsymbol{\omega})\boldsymbol{R}(\boldsymbol{v})\boldsymbol{h} + \boldsymbol{B}\boldsymbol{u}\end{pmatrix}.$$

Here $\boldsymbol{E}(\boldsymbol{v}), \boldsymbol{S}(\boldsymbol{\omega}), \boldsymbol{R}(\boldsymbol{v}) : \mathbb{R}^3 \to \mathbb{R}^{3\times 3}$ are matrix-valued functions defined as

$$\boldsymbol{E}(\boldsymbol{v}) \coloneqq \begin{pmatrix}1 & \sin\phi\tan\theta & \cos\phi\tan\theta \\ 0 & \cos\phi & -\sin\phi \\ 0 & \sin\phi/\cos\theta & \cos\phi/\cos\theta\end{pmatrix}, \qquad \boldsymbol{S}(\boldsymbol{\omega}) \coloneqq \begin{pmatrix}0 & \omega_3 & -\omega_2 \\ -\omega_3 & 0 & \omega_1 \\ \omega_2 & -\omega_1 & 0\end{pmatrix},$$

and

$$\boldsymbol{R}(\boldsymbol{v}) := \begin{pmatrix} \cos\theta\cos\psi & \cos\theta\sin\psi & -\sin\theta \\ \sin\phi\sin\theta\cos\psi - \cos\phi\sin\psi & \sin\phi\sin\theta\sin\psi + \cos\phi\cos\psi & \cos\theta\sin\phi \\ \cos\phi\sin\theta\cos\psi + \sin\phi\sin\psi & \cos\phi\sin\theta\sin\psi - \sin\phi\cos\psi & \cos\theta\cos\phi \end{pmatrix}.$$

Further, $\boldsymbol{J} \in \mathbb{R}^{3\times 3}$ is a combination of the inertia matrices of the momentum wheels and the rigid body without wheels, $\boldsymbol{h} \in \mathbb{R}^3$ is the total constant angular momentum of the system, and $\boldsymbol{B} \in \mathbb{R}^{3\times m}$ is a constant matrix where m is the number of momentum wheels. To control the system, we apply a torque $\boldsymbol{u}(t, \boldsymbol{v}, \boldsymbol{\omega}) : [0, t_f] \times \mathbb{R}^3 \times \mathbb{R}^3 \to \mathbb{R}^m$.

We consider the fully-actuated case where $m = 3$. Let

$$\boldsymbol{B} = \begin{pmatrix} 1 & 1/20 & 1/10 \\ 1/15 & 1 & 1/10 \\ 1/10 & 1/15 & 1 \end{pmatrix}, \qquad \boldsymbol{J} = \begin{pmatrix} 2 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 4 \end{pmatrix}, \qquad \boldsymbol{h} = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}.$$

The optimal control problem is

$$\text{(5.1)} \qquad \begin{cases} \underset{\boldsymbol{u}(\cdot)}{\text{minimize}} & J\left[\boldsymbol{u}(\cdot)\right] = \displaystyle\int_t^{t_f} \mathcal{L}(\boldsymbol{v}, \boldsymbol{\omega}, \boldsymbol{u}) d\tau + \frac{W_4}{2}\|\boldsymbol{v}(t_f)\|^2 + \frac{W_5}{2}\|\boldsymbol{\omega}(t_f)\|^2, \\ \text{subject to} & \dot{\boldsymbol{v}} = \boldsymbol{E}(\boldsymbol{v})\boldsymbol{\omega}, \\ & \boldsymbol{J}\dot{\boldsymbol{\omega}} = \boldsymbol{S}(\boldsymbol{\omega})\boldsymbol{R}(\boldsymbol{v})\boldsymbol{h} + \boldsymbol{B}\boldsymbol{u}. \end{cases}$$

Here

$$\mathcal{L}(\boldsymbol{v}, \boldsymbol{\omega}, \boldsymbol{u}) = \frac{W_1}{2}\|\boldsymbol{v}\|^2 + \frac{W_2}{2}\|\boldsymbol{\omega}\|^2 + \frac{W_3}{2}\|\boldsymbol{u}\|^2$$

and

$$W_1 = 1, \quad W_2 = 10, \quad W_3 = \frac{1}{2}, \quad W_4 = 1, \quad W_5 = 1, \quad t_f = 20.$$

Finally, we consider initial conditions in the domain

$$\text{(5.2)} \qquad \mathbb{X}_0 = \left\{ \boldsymbol{v}, \boldsymbol{\omega} \in \mathbb{R}^3 \middle| -\frac{\pi}{3} \le \phi, \theta, \psi \le \frac{\pi}{3} \text{ and } -\frac{\pi}{4} \le \omega_1, \omega_2, \omega_3 \le \frac{\pi}{4} \right\}.$$

In [25], to avoid discretizing time the value function is approximated only at initial time $t = 0$. In order to facilitate a fair comparison we do the same. This means that we model $V(0, \boldsymbol{v}, \boldsymbol{\omega}) \approx V^{\text{NN}}(\boldsymbol{v}, \boldsymbol{\omega})$, i.e. the NN does *not* take time as an input variable. Consequently the control is implemented with a time-independent moving horizon rather than as a time-dependent optimal control. In other words, at each time t when we evaluate the control, we assume $t = 0$ and return $\boldsymbol{u}(t) = \boldsymbol{u}^{\text{NN}}(\boldsymbol{v}(t), \boldsymbol{\omega}(t))$. Controlling the system using moving horizon feedback is standard practice. It is also reasonable for the present case because the problem dynamics are time-invariant and the time horizon is relatively long. Because of this we observe near-optimal performance from the moving horizon controller.

5.1. Learning the value function. In this section, we present numerical results of our implementation of a NN for modeling the initial-time value function of the rigid body attitude control problem (5.1). To obtain data, we uniformly sample initial conditions $\left(\boldsymbol{v}^{(i)}, \boldsymbol{\omega}^{(i)}\right)$ from the domain $\mathbb{X}_0$ defined in (5.2), and for each initial value, solve the two-point BVP (2.11) using time-marching and the SciPy [37] implementation of the three-stage Lobatto IIIa algorithm in [26]. Each integrated trajectory contains around 100 data points on average, but we use only initial time data, $V\left(0, \boldsymbol{v}^{(i)}, \boldsymbol{\omega}^{(i)}\right)$. For validation, we generate a data set containing $|\mathcal{D}_{\text{val}}| = 1000$

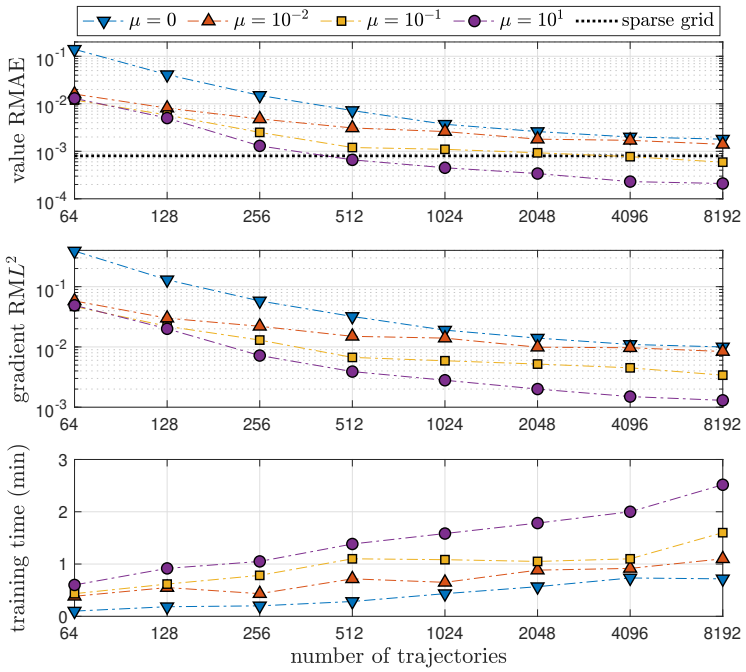


Figure 1: Validation accuracy and training time of NNs for modeling the initial time value function $V(0, \boldsymbol{v}, \boldsymbol{\omega})$ of the optimal attitude control problem (5.1). All NNs have the same parameter initialization and are run on an NVIDIA RTX 2080Ti GPU.

data points (at $t = 0$), and keep this fixed throughout all the tests. As a baseline, the sparse grid characteristics method with $\left|G^{13}_{\text{sparse}}\right| = 44,698$ grid points achieves a RMAE of 8.00×10^{-4} on this validation data set.

We implement a standard feedforward NN in TensorFlow 1.11 [1] and train it to approximate $V(0, \boldsymbol{v}, \boldsymbol{\omega})$. The NN has three hidden layers with 64 neurons in each, but many alternate configurations of depth and width also work. For optimization, we use the SciPy interface for the L-BFGS optimizer [37, 8]. Figure 1 displays the results of a series of tests in which we vary the weight μ on the value gradient loss term (3.5) and the size of the training data set. Results are compared to those obtained in [25].

We highlight that with just 512 data points, we can train NNs with better accuracy than the sparse grid characteristics method with $\left|G^{13}_{\text{sparse}}\right| = 44,698$ points. Thus for this problem, the proposed method is about 90 times as data-efficient. With 8192 data points, the NN can be almost four times as accurate as the sparse grid characteristics method. This level of accuracy with small data sets is obtained only with physics-informed learning. In particular, NNs trained by pure regression (3.1) cannot match the accuracy of the sparse grid characteristics method, as shown in Figure 1 for the case with $\mu = 0$. Accuracy improves as we increase μ but with diminishing returns for $\mu \geq 10$. While physics-informed learning is more costly, it permits the use of much smaller data sets, and the increased training time is still quite short.

5.2. Training with adaptive data generation. Performing a thorough systematic study of the adaptive sampling and model refinement technique proposed in

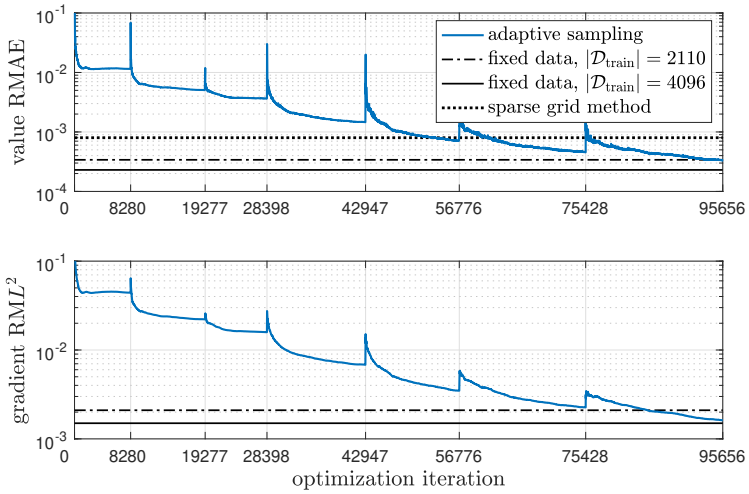


Figure 2: Progress of adaptive sampling and model refinement for the rigid body problem (5.1), compared to training on fixed data sets and the sparse grid characteristics method. Spikes in the error correspond to the start of new training rounds and expansion of the training data set.

section 4 is rather complicated, since a successful implementation depends on various hyperparameter settings, which can and perhaps should change each optimization round. Results also depend on random chance, since data points are chosen in a (partially) random way and the randomly-initialized NN training problem is highly non-convex. For this reason, in this section we show a just few conservative results which we feel illustrate the potential of the method.

Figure 2 shows the progress of the validation error during training when using adaptive sampling starting from a data set with $\left|\mathcal{D}^1_{\text{train}}\right| = 64$ points. We set the gradient loss weight to $\mu = 10$ and the convergence parameter in (4.8) to $C = 0.25$. After each round, we check the convergence criterion (4.8) and increase the number of training data according to (4.9). Each data set includes all previously generated data, and we generate additional data as needed through Algorithm 4.1. With these configurations, the model passes the convergence test after seven training rounds and observing a total of $\left|\mathcal{D}^7_{\text{train}}\right| = 2110$ samples.

The final value function accuracy is 3.3×10^{-4}: over twice as accurate as the sparse grid method with about twenty times fewer data, and the gradient prediction accuracy is 1.6×10^{-3}. As shown in Figure 2, the gradient predictions of the network trained using the adaptive algorithm are just as accurate as a network trained on a fixed data set of $|\mathcal{D}_{\text{train}}| = 4096$ samples. That is to say, the adaptive sampling method facilitates more acurate gradient predictions using fewer data. These results highlight the main advantages of the adaptive sampling and model refinement method: the ability to overcome an initial lack of data, efficiently generate a large data set, and improve gradient prediction accuracy which is needed for effective control. To fully realize the potential of the method, hyperparameters like μ, C, and internal optimizer parameters need to be adjusted in each round. Development of algorithms to do this adaptively remains a topic for future research.

K	% BVP convergence	mean integration time
1	0.3%	0.37 s
2	38.7%	0.44 s
3	76.2%	0.40 s
4	92.9%	0.45 s
8	98.4%	0.53 s

Table 1: Convergence of BVP solutions for (5.1) when using the time-marching trick, depending on the number of steps in the sequence $\{t_k\}_{k=1}^K$. The case $K = 1$ corresponds to a direct solution attempt over the whole time interval with no time-marching. BVP integration time is measured only on successful attempts – failed solution attempts usually take much longer.

μ	training time	gradient RML^2	% BVP conv.	mean int. time
10^{-8}	7 s	2.5×10^{-1}	88.0%	0.50 s
10^{-4}	19 s	1.4×10^{-1}	98.6%	0.48 s
1	23 s	4.5×10^{-2}	99.7%	0.44 s

Table 2: Convergence of BVP solutions for (5.1) when using NN warm start with NNs of varying gradient prediction accuracy. BVP integration time is measured only on successful attempts.

Next, we investigate the convergence of the BVP solver with time-marching and NN warm start. Results are given in Table 1 and Table 2, respectively. For these tests, we use 1000 initial conditions with the largest predicted gradient norm, $\|V_{\boldsymbol{x}}^{\mathrm{NN}}(\cdot)\|$, picked from a set of 10^6 randomly sampled candidate points. Initial conditions with large gradient norm tend to be located in regions where the value function is steep or complicated, and may thus be more difficult to solve. The set of initial conditions is fixed for all tests.

In the first row of Table 1, we attempt to solve the BVP with no time-marching, i.e. over the entire time interval without constructing any initial guess. In this case, the proportion of convergent solutions is extremely small, obviating the need for good initial guesses. As shown in Table 1, we reliably obtain solutions for this problem when we use at least $K = 4$ time intervals. We note that the initial conditions are purposefully chosen to be difficult – if we simply take uniform samples from the domain $\mathbb{X}_0$, the proportion of convergent solutions increases significantly.

In Table 2, we present results using NN warm start. We train several NNs on a data set of only 64 points. Because the data set is so small, *each NN takes only seconds to finish training.* We also experiment with using different gradient loss weights μ for each NN. This directly impacts the accuracy in predicting the initial-time costate, $\boldsymbol{\lambda}(0; \boldsymbol{v}_0, \boldsymbol{\omega}_0) \approx V_{\boldsymbol{x}}^{\mathrm{NN}}(\boldsymbol{v}_0, \boldsymbol{\omega}_0)$, which in turn is key to synthezing optimal controls.

Even with these low-fidelity models, the rate of BVP convergence is just as high as when using $K = 4$ time intervals for time-marching. The quality of initial guesses improves with better costate prediction, and it is not difficult to exceed 99% convergence. For this problem, the speed of the two methods is about the same. However, when we consider higher-dimensional problems in subsection 6.2, we find that NN warm start significantly improves both reliability and efficiency.

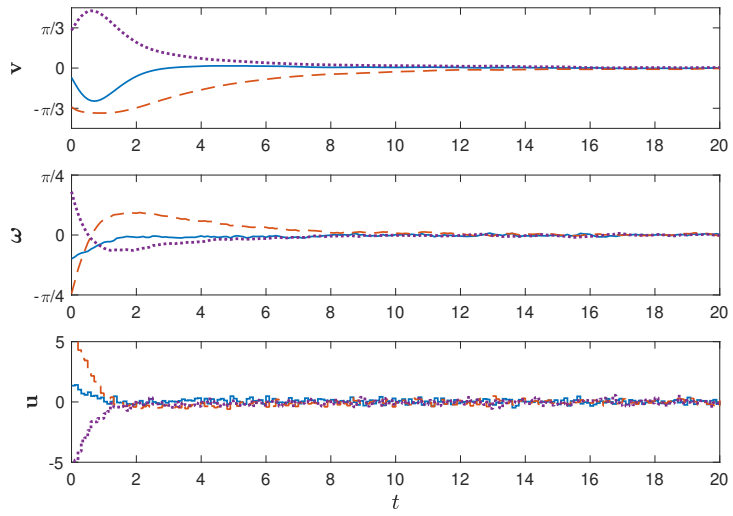


Figure 3: Sample closed-loop trajectories of the rigid body system controlled by a NN feedback controller implemented with a zero-order-hold and subject to measurement noise. Solid: ϕ, ω_1, and u_1. Dashed: θ, ω_2, and u_2. Dotted: ψ, ω_3, and u_3.

5.3. Closed-loop simulation. In this section we perform numerical simulations of the rigid body dynamics, demonstrating that the NN feedback controller is capable of stabilizing the system. Using (3.9) we calculate the optimal feedback control law

$$\boldsymbol{u}^{\mathrm{NN}}(\boldsymbol{v},\boldsymbol{\omega}) = -\frac{1}{W_3}\left[\boldsymbol{J}^{-1}\boldsymbol{B}\right]^T V_{\boldsymbol{\omega}}^{\mathrm{NN}}(\boldsymbol{v},\boldsymbol{\omega}). \tag{5.3}$$

Recall that because we are using a time-independent value function model, the control is implemented as time-independent moving horizon feedback. Since $\boldsymbol{J}$ and $\boldsymbol{B}$ are constant matrices, we pre-compute the product $-\left[\boldsymbol{J}^{-1}\boldsymbol{B}\right]^T/W_3$. Hence evaluation of the control requires only a forward pass through the computational graph of $V_{\boldsymbol{\omega}}^{\mathrm{NN}}(\cdot)$ and a matrix multiplication.

In Figure 3, we plot a typical closed-loop trajectory starting from a randomly sampled initial condition. To make the simulation more realistic, we implement the controller using a zero-order-hold with a sample rate of 10 [Hz]. In addition, we corrupt inputs to the controller with Gaussian white noise with standard deviation $\sigma = 0.01\pi$. That is, for all $t \in [t_k, t_k + 0.1]$, we apply the control

$$\boldsymbol{u}(t) = \boldsymbol{u}^{\mathrm{NN}}\left(\hat{\boldsymbol{v}}(t_k), \hat{\boldsymbol{\omega}}(t_k)\right),$$

where

$$\begin{pmatrix}\hat{\boldsymbol{v}}(t_k)\\ \hat{\boldsymbol{\omega}}(t_k)\end{pmatrix} \coloneqq \begin{pmatrix}\boldsymbol{v}(t_k)\\ \boldsymbol{\omega}(t_k)\end{pmatrix} + \boldsymbol{n}(t_k), \qquad \boldsymbol{n}(t_k) \sim \mathcal{N}\left(\boldsymbol{0}, \sigma^2\boldsymbol{I}\right).$$

In spite of this, the NN controller successfully stabilizes the system. Furthermore, the total cost of the closed-loop trajectory is $J\left[\boldsymbol{u}^{\mathrm{NN}}(\cdot)\right] = 12.67$, about 1% more than the optimal cost $J\left[\boldsymbol{u}^{*}(\cdot)\right] = 12.52$. For comparison, a linear quadratic regulator (LQR) for (5.1) accumulates a total cost of $J\left[\boldsymbol{u}^{\mathrm{LQR}}(\cdot)\right] = 15.95$, which is 27% more than

the optimal cost. Finally, short computation time is critical for implementation in real systems, and this is achieved here as each evaluation of the control takes only a couple milliseconds on both an NVIDIA RTX 2080Ti GPU and a 2012 MacBook Pro.

6. Application to control of Burgers'-type PDE. In this section, we test our method on high-dimensional nonlinear systems arising from a Chebyshev pseudospectral (PS) discretization of a one-dimensional forced Burgers'-type PDE. An infinite-horizon version of this problem is studied in [22], in which the value function is approximated using a polynomial Galerkin technique. We note that in [22], separability of the nonlinear dynamics is required to compute the high-dimensional integrals necessary in the Galerkin formulation. Our method does not require this restriction, although it does apply to this problem.

As in [22], let $X(t,\xi) : [0,t_f] \times [-1,1] \to \mathbb{R}$ satisfy the following one-dimensional controlled PDE with Dirichlet boundary conditions:

$$\begin{cases} X_t = XX_\xi + \nu X_{\xi\xi} + \alpha X e^{\beta X} + I_\Omega(\xi)u, & t > 0, \xi \in (-1,1), \\ X(t,-1) = X(t,1) = 0, & t > 0, \\ X(0,\xi) = X_0, & \xi \in (-1,1). \end{cases} \tag{6.1}$$

For notational convenience we have written $X = X(t,\xi)$, and as before we denote $X_t = \partial X/\partial t$, $X_\xi = \partial X/\partial \xi$, and $X_{\xi\xi} = \partial^2 X/\partial \xi^2$. The scalar-valued control $u(t,X)$ is actuated only on Ω, the support of the indicator function

$$I_\Omega(\xi) \coloneqq \begin{cases} 1, & \xi \in \Omega, \\ 0, & \xi \notin \Omega\ . \end{cases}$$

The PDE-constrained optimal control problem is

$$\begin{cases} \underset{u(\cdot)}{\text{minimize}} & J\left[u(\cdot)\right] = \displaystyle\int_t^{t_f} \mathcal{L}(X,u)d\tau + \frac{W_2}{2}\left\|X(t_f,\xi)\right\|^2_{L^2_{(-1,1)}}, \\ \text{subject to} & X_t = XX_\xi + \nu X_{\xi\xi} + \alpha X e^{\beta X} + I_\Omega(\xi)u, \\ & X(\tau,-1) = X(\tau,1) = 0. \end{cases} \tag{6.2}$$

Here

$$\left\|X(\tau,\xi)\right\|^2_{L^2_{(-1,1)}} \coloneqq \int_{-1}^{1} X^2(\tau,\xi)d\xi, \qquad \mathcal{L}(X,u) = \frac{1}{2}\left\|X(\tau,\xi)\right\|^2_{L^2_{(-1,1)}} + \frac{W_1}{2}u^2(\tau,X),$$

and we set

$$\Omega = (-0.5,-0.2), \quad \nu = 0.2, \quad \alpha = 1.5, \quad \beta = -0.1, \quad W_1 = 0.1, \quad W_2 = 1, \quad t_f = 8.$$

In this problem, the goal of stabilizing $X(t,\xi)$ is made more challenging by the added reaction term, $\alpha X e^{\beta X}$, which renders the origin unstable. This can be seen clearly in Figure 4a.

To solve (6.2) using our framework, we perform Chebyshev PS collocation to transform the PDE (6.1) into a system of ordinary differential equations (ODEs). Following [36], let

$$\xi_j = \cos(j\pi/N_c), \qquad j = 0,1,\dots N_c,$$

where $N_c + 1$ is the number of collocation points. After accounting for boundary conditions, we collocate $X(t,\xi)$ at internal (non-boundary) Chebyshev points, ξ_j, $j = 1,2,\dots,n$, where $n = N_c - 1$. The discretized state is defined as

$$\boldsymbol{x}(t) \coloneqq \begin{pmatrix} X(t,\xi_1), & X(t,\xi_2), & \dots, & X(t,\xi_n) \end{pmatrix}^T : [0,t_f] \to \mathbb{R}^n,$$

and the PDE (6.1) becomes a system of ODEs in n dimensions:

$$\dot{\boldsymbol{x}} = \boldsymbol{x} \odot \boldsymbol{D}\boldsymbol{x} + \nu \boldsymbol{D}^2 \boldsymbol{x} + \alpha \boldsymbol{x} \odot e^{\beta \boldsymbol{x}} + \mathbb{I}_\Omega u,$$

In the above, "$\odot$" denotes element-wise multiplication (Hadamard product), $\mathbb{I}_\Omega$ is the discretized indicator function, and $\boldsymbol{D}, \boldsymbol{D}^2 \in \mathbb{R}^{n \times n}$ are the internal parts of the first and second order Chebyshev differentiation matrices, which are obtained by deleting the first and last rows and columns of the full matrices. This discretization automatically enforces the boundary conditions. Finally, since $X(t, \xi)$ is collocated at Chebyshev nodes, the inner product appearing in the cost function is conveniently approximated by Clenshaw-Curtis quadrature [36]:

$$\|X(\tau, \xi)\|^2_{L^2_{(-1,1)}} = \int_{-1}^{1} X^2(\tau, \xi) d\xi \approx \boldsymbol{w}^T \boldsymbol{x}^2(\tau),$$

where $\boldsymbol{w} \in \mathbb{R}^n$ are the internal Clenshaw-Curtis quadrature weights and $\boldsymbol{x}^2(t)$ is calculated element-wise, i.e. $\boldsymbol{x}^2 = \boldsymbol{x} \odot \boldsymbol{x}$. Now the original OCP (6.2) can be reformulated as an ODE-constrained problem,

$$\text{(6.3)} \qquad \begin{cases} \underset{u(\cdot)}{\text{minimize}} & \displaystyle\int_t^{t_f} \frac{1}{2} \left[\boldsymbol{w}^T \boldsymbol{x}^2(\tau) + W_1 u^2(\tau, \boldsymbol{x}) \right] d\tau + \frac{W_2}{2} \boldsymbol{w}^T \boldsymbol{x}^2(t_f), \\ \text{subject to} & \dot{\boldsymbol{x}} = \boldsymbol{x} \odot \boldsymbol{D}\boldsymbol{x} + \nu \boldsymbol{D}^2 \boldsymbol{x} + \alpha \boldsymbol{x} \odot e^{\beta \boldsymbol{x}} + \mathbb{I}_\Omega u. \end{cases}$$

6.1. Learning high-dimensional value functions. The state dimension n of the OCP (6.3) can be adjusted, presenting a good opportunity to test the scalability of our algorithms. For this problem, we learn the value function $V = V(t, \boldsymbol{x})$ with time-dependence, rather than just $V(0, \boldsymbol{x})$ as in section 5. Consequently, the resulting controls can be implemented as time-dependent controls or with a moving horizon. We consider the following domain of initial conditions:

$$\text{(6.4)} \qquad \mathbb{X}_0 = \{ \boldsymbol{x} \in \mathbb{R}^n | -2 \leq x_j \leq 2, j = 1, 2, \ldots, n \}.$$

Using the proposed adaptive deep learning framework, we approximate solutions to (6.3) in $n = 10$, 20, and 30 dimensions. We focus on demonstrating what is possible using our approach, rather than carrying out a detailed study of its effectiveness under different parameter tunings. In [22] an infinite-horizon version of the problem is solved up to twelve dimensions, but the accuracy of the solution is not readily verifiable. The ability to conveniently measure model accuracy for general high-dimensional problems with *no known analytical solution* is a key advantage of our framework.

For each discretized OCP, $n = 10$, 20, and 30, we apply the time-marching strategy to build an initial training data set $\mathcal{D}^1_{\text{train}}$ from 30 uniformly sampled initial conditions, $\boldsymbol{x}_0^{(i)} \in \mathbb{X}_0$, $i = 1, 2, \ldots, 30$. For each initial condition $\boldsymbol{x}_0^{(i)}$, the BVP solver outputs an optimal trajectory $\left\{ \boldsymbol{x}^{(i)}(t_k) \right\}$, evaluated at collocation points $t_k \in [0, t_f]$ chosen by the solver. Typically this can be a few hundred per initial condition, depending on the state dimension n and the BVP solver tolerances. Since these data sets can be get quite large, we often train on randomly selected subsets of the data. This can significantly improve training speed without sacrificing accuracy. When neeeded, we solve additional BVPs to expand the data set as described in subsection 4.2. We use the same NN architecture as in section 5, with three hidden layers with 64 neurons each. We set $C = 0.3$, 1.3, and 1.8 for $n = 10$, 20, and 30, respectively, and use $\mu = 10$ in all cases.

n	num. trajectories	training time	value RMAE	gradient RML^2
10	163	25 min	1.3×10^{-3}	5.7×10^{-3}
20	128	48 min	5.0×10^{-3}	1.1×10^{-2}
30	145	62 min	1.3×10^{-2}	2.2×10^{-2}

Table 3: Validation accuracy of NNs for solving the collocated Burgers'-type OCP (6.3), depending on the state dimension n. Training time includes time spent generating additional data according to Algorithm 4.1. All NNs are trained on an NVIDIA RTX 2080Ti GPU.

In Table 3, we present validation accuracy results for the trained NNs. We include the RMAE in predicting the value function and the RML^2 error in predicting the costate, $\boldsymbol{\lambda}(t;\boldsymbol{x}_0) \approx V_{\boldsymbol{x}}^{\mathrm{NN}}(t, \boldsymbol{x}(t;\boldsymbol{x}_0))$. Accuracy is measured empirically on independently generated validation data sets comprised of trajectories from 50 randomly selected initial conditions. We find that the trained NNs accurately predict both the value function and its gradient, even in 30 dimensions.

Table 3 also shows the total number of sample trajectories seen by the NN, including the initial data $\mathcal{D}^1_{\mathrm{train}}$. It may seem surprising that we are able to reach the same level of accuracy in higher dimensions with similar numbers of sample trajectories. This happens because the BVP solver usually needs more collocation points for larger problems, thus producing more data per trajectory. Consequently, fewer trajectories are needed to fulfill the data set size recommendation (4.9). Similarly, in section 5 we use data only for $t = 0$, so we need thousands of trajectories to fill in the state space and train the NN. This suggests that learning the time-dependent value function can be more efficient than learning $V(0, \boldsymbol{x})$ only. Note that, if preferred, the time-dependent controller can still be used with a moving horizon like in section 5. Such an implementation can be useful in the presence of noise.

Lastly, Table 3 shows the training time for each NN, including time spent testing convergence and generating additional trajectories on the fly, but not time spent generating the initial data. Generating data becomes the most expensive computation as n increases, but even so we find that computational effort scales reasonably with the problem dimension. Furthermore, it is possible to obtain a rough low-fidelity NN model in just minutes as shown in Table 5, which in turn allows for more efficient data generation. This demonstrates the viability of the proposed method for solving high-dimensional optimal control problems.

6.2. NN warm start for fast and reliable BVP solutions. In our experience, generating the initial training data set can be the most computationally demanding part of the process, especially as the problem dimension n increases. Consequently, for difficult high-dimensional problems it may be impractical to generate a large-enough data set from scratch. This obstacle can be largely overcome by using partially-trained/low-fidelity NNs to aid in further data generation. In this section, we briefly compare the reliability and speed of BVP convergence between our two strategies: time-marching and NN warm start. These experiments demonstrate the importance of NN guesses for high-dimensional data generation.

For each of $n = 10$, 20, and 30, we randomly sample a set of 1000 candidate points from the domain $\mathbb{X}_0$ defined in (6.4). From these we choose 100 points with the largest predicted value gradient. The set of initial conditions is fixed for each n. Next we

n	K	% BVP convergence	mean integration time
10	4	40%	0.7 s
	6	83%	0.8 s
	10	90%	1.3 s
20	4	46%	3.6 s
	5	86%	4.2 s
	6	99%	4.7 s
30	4	47%	11.3 s
	6	90%	14.6 s
	8	100%	19.1 s

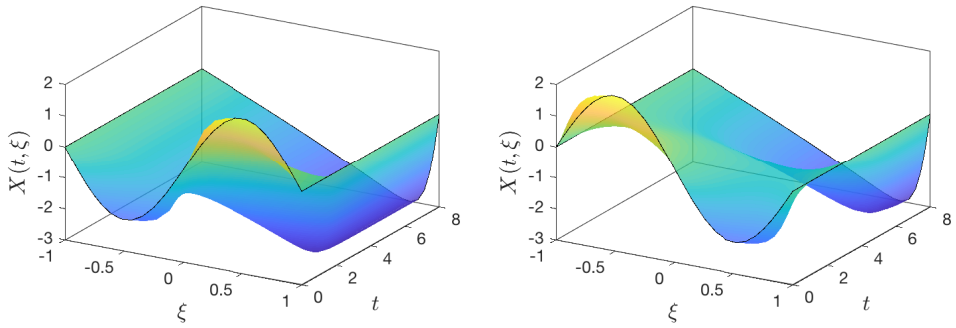
Table 4: Convergence of BVP solutions for (6.3) when using the time-marching trick, depending on the problem dimension, n, and the number of steps in the sequence $\{t_k\}_{k=1}^K$. BVP integration time is measured only on successful attempts.

n	μ	training time	gradient RML^2	% BVP conv.	mean int. time
10	10^{-8}	20 s	2.1×10^{-1}	96%	0.8 s
	10^{-4}	31 s	9.8×10^{-2}	99%	0.8 s
	1	56 s	4.2×10^{-2}	88%	0.6 s
20	10^{-8}	29 s	3.7×10^{-1}	74%	2.9 s
	10^{-4}	47 s	9.8×10^{-2}	91%	2.5 s
	1	76 s	6.5×10^{-2}	98%	2.5 s
30	10^{-8}	38 s	3.0×10^{-1}	79%	7.1 s
	10^{-4}	125 s	7.6×10^{-2}	94%	6.9 s
	1	189 s	7.4×10^{-2}	96%	7.1 s

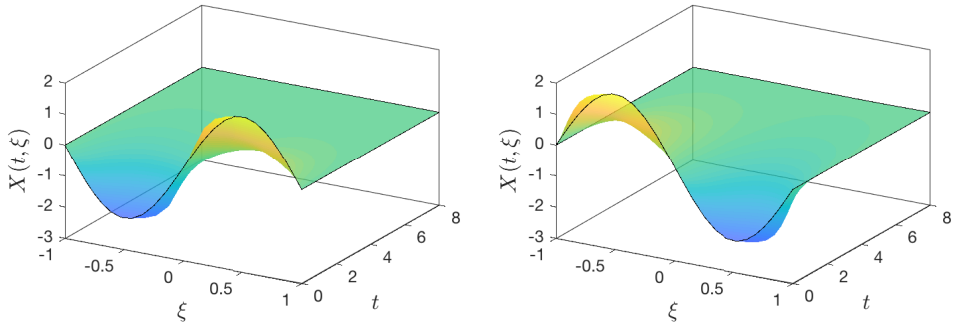
Table 5: Convergence of BVP solutions for (6.3) when using NN warm start with NNs of varying gradient prediction accuracy. BVP integration time is measured only on successful attempts.

proceed as in subsection 5.2, solving each BVP by time-marching with various K. Results are summarized in Table 4. We then solve the same BVPs directly over the whole time interval $t \in [0, 8]$ with NN warm start. These NNs are trained on fixed data sets containing only 30 trajectories, but with different gradient loss weights μ, resulting in varying costate prediction accuracy. We also limit the number of L-BFGS iterations so that each model is trained only for a short time. Results are given in Table 5.

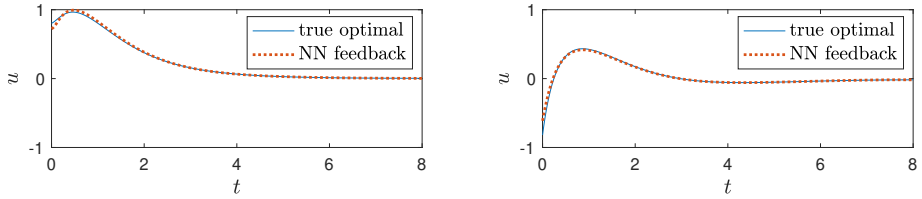
As before, we find that even NNs with relatively large costate prediction error enable consistently convergent BVP solutions. Time-marching also works once the sequence of time steps $\{t_k\}_{k=1}^K$ is properly tuned, but the speed of this method scales poorly with n. Now the advantage of utilizing NNs to aid in data generation becomes clear: when n is large, the average time needed for convergence when using NN warm start is drastically lower than that of the time-marching trick. This approach also requires no tuning of the time-marching sequence. Because low-fidelity NNs are quick to train, training such a NN and then using it to aid in data generation is the most efficient strategy for building larger data sets.



(a) Uncontrolled dynamics.



(b) NN-controlled dynamics.



(c) Comparison of true optimal control (open-loop BVP solution) and NN control profiles.

Figure 4: Simulations of the collocated Burgers'-type PDE (6.1) in $n = 30$ dimensions. Left column: $X(0,\xi) = 2\sin(\pi\xi)$. Right column: $X(0,\xi) = -2\sin(\pi\xi)$.

6.3. Closed-loop simulations. In this section we use simulations to demonstrate that the feedback control output by the trained NN not only stabilizes the high-dimensional system, but that it is close to the true optimal control. The optimal feedback control law can again be calculated with (3.9), from which we obtain

$$u^{\mathrm{NN}}(t,\boldsymbol{x}) = -\frac{1}{W_1}\mathbb{I}_\Omega^T V_{\boldsymbol{x}}^{\mathrm{NN}}(t,\boldsymbol{x}). \tag{6.5}$$

In Figure 4, we plot the uncontrolled (Figure 4a) and closed-loop controlled dynamics (Figure 4b), starting from two different initial conditions, $X(0,\xi) = 2\sin(\pi\xi)$

and $X(0,\xi) = -2\sin(\pi\xi)$, where the dimension of the discretized system is $n = 30$. For both of these initial conditions (and almost all others tested), the NN controller successfully stabilizes the open-loop unstable origin. Further, as shown in Figure 4c, the NN-generated controls are very close to the true optimal controls which are calculated by solving the associated BVPs. Finally, the speed of online control computation is not sensitive to the problem dimension: each evaluation still takes just milliseconds on both an NVIDIA RTX 2080Ti GPU and a 2012 MacBook Pro.

7. Conclusion. In this paper, we have developed a novel machine learning framework for solving HJB equations and designing candidate optimal feedback controllers. Unlike many other state of the art techniques, our method does not require finite difference approximations of the gradient nor strict restrictions on the structure of the dynamics. The causality-free algorithm we use for data generation enables application to high-dimensional systems and validation of model accuracy. We also emphasize that while our method is data-driven, by leveraging the costate data we are able to train more physically-consistent models and better controllers with surprisingly small data sets.

The proposed method is not only a consumer of data, but through adaptive data generation it can also be used build rich data sets with points anywhere in a semi-global domain. Thus the value function and control are valid for large ranges of dynamic states, rather than just in the neighborhood of some nominal trajectory. Furthermore, data can be generated near complicated or non-smooth regions of the value function to aid in learning. This in turn allows us to train more accurate NN models or employ other data-driven methods.

We have demonstrated the possibility for use of the framework in a practical setting by synthesizing candidate optimal feedback controls of a six-dimensional nonlinear rigid body. The potential for scalability of the method is demonstrated by solving HJB equations in up to 30 dimensions using limited data, and empirical validation indicates that the NN models are good approximations of the value function. How well the proposed techniques work for even larger problems remains an open question. Indeed, understanding the scalability of deep learning methods in general is still an active area of research. Nevertheless, we are encouraged by the simulations in section 6 which suggest that the method may scale quite well for moderately high-dimensional problems. In addition, the computational burden associated with an increase in dimensionality is incurred entirely offline: due to the structure of NNs, increasing the dimension has a negligible impact on the speed of online control calculation.

These promising results leave plenty of room for future development. Of special interest are extensions of the framework to solve problems with free final time and state and control constraints, which appear ubiquitously in practical applications. Such problems typically give rise to non-unique solutions of PMP and non-smooth value functions, thus presenting substantial challenges for both data generation and neural network modeling. Overcoming these obstacles would open the door to solving many important and difficult optimal control problems.

REFERENCES

[1] M. Abadi, A. Agarwal, P. Barham, et al., *TensorFlow: Large-scale machine learning on heterogeneous systems*, 2016, https://arxiv.org/abs/1603.04467.

[2] M. Abu-Khalaf and F. L. Lewis, *Nearly optimal control laws for nonlinear systems with saturating actuators using a neural network HJB approach*, Automatica, 41 (2005), pp. 779–

791, https://doi.org/10.1016/j.automatica.2004.11.034.
[3] E. Al'brekht, *On the optimal stabilization of nonlinear systems*, J. Appl. Math. Mech., 25(5) (1961), pp. 1254–1266, https://doi.org/10.1016/0021-8928(61)90005-3.
[4] A. Bachouch, C. Huré, N. Langrené, and H. Pham, *Deep neural networks algorithms for stochastic control problems on finite horizon: numerical applications*, 2018, https://arxiv.org/abs/1812.05916.
[5] O. Bokanowski, J. Garcke, M. Griebel, and I. Klompmaker, *An adaptive sparse grid semi-Lagrangian scheme for first order Hamilton-Jacobi Bellman equations*, J. Sci. Comput., 55 (2013), pp. 575–605, https://doi.org/10.1007/s10915-012-9648-x.
[6] L. Bottou, F. E. Curtis, and J. Nocedal, *Optimization methods for large-scale machine learning*, SIAM Rev., 60 (2018), pp. 223–311, https://doi.org/10.1137/16M1080173.
[7] R. H. Byrd, G. M. Chin, J. Nocedal, and Y. Wu, *Sample size selection in optimization methods for machine learning*, Math. Program., 134 (2012), pp. 127–155, https://doi.org/10.1007/s10107-012-0572-5.
[8] R. H. Byrd, P. Lu, J. Nocedal, and C. Zhu, *A limited memory algorithm for bound constrained optimization*, SIAM J. Sci. Comput., 16 (1995), pp. 1190–1208, https://doi.org/10.1137/0916069.
[9] S. Cacace, E. Cristiani, M. Falcone, and A. Picarelli, *A patchy dynamic programming scheme for a class of Hamilton-Jacobi-Bellman equations*, SIAM J. Sci. Comput., 34 (2012), pp. A2625–A2649, https://doi.org/10.1137/110841576.
[10] T. Cheng, F. L. Lewis, and M. Abu-Khalaf, *Fixed-final-time-constrained optimal control of nonlinear systems using neural network HJB approach*, IEEE Trans. Neural Netw., 18 (2007), pp. 1725–1737, https://doi.org/10.1109/TNN.2007.905848.
[11] Y. T. Chow, J. Darbon, S. Osher, and W. Yin, *Algorithm for overcoming the curse of dimensionality for state-dependent Hamilton-Jacobi equations*, J. Comput. Phys., 387 (2019), pp. 376–409, https://doi.org/10.1016/j.jcp.2019.01.051.
[12] M. G. Crandall and P.-L. Lions, *Viscosity solutions of Hamilton-Jacobi equations*, Trans. Amer. Math. Soc., 277 (1983), pp. 1–42, https://doi.org/10.2307/1999343.
[13] J. Darbon, G. P. Langlois, and T. Meng, *Overcoming the curse of dimensionality for some Hamilton–Jacobi partial differential equations via neural network architectures*, Res. Math. Sci., 7 (2020), p. 20, https://doi.org/10.1007/s40687-020-00215-6.
[14] J. Darbon and T. Meng, *On some neural network architectures that can represent viscosity solutions of certain high dimensional Hamilton–Jacobi partial differential equations*, J. Comput. Phys., 425 (2021), p. 109907, https://doi.org/https://doi.org/10.1016/j.jcp.2020.109907.
[15] J. Darbon and S. Osher, *Algorithms for overcoming the curse of dimensionality for certain Hamilton-Jacobi equations arising in control theory and elsewhere*, Res. Math. Sci., 3 (2016), https://doi.org/10.1186/s40687-016-0068-7.
[16] J. Diebel, *Representing attitude: Euler angles, unit quaternions, and rotation vectors*, 2006, https://www.astro.rug.nl/software/kapteyn-beta/_downloads/attitude.pdf (accessed 2020-05-16).
[17] M. Falcone and R. Ferretti, *Semi-Lagrangian Approximation Schemes for Linear and Hamilton-Jacobi Equations*, Society for Industrial and Applied Mathematics, Philadelphia, PA, 2013, https://doi.org/10.1137/1.9781611973051.
[18] J. Han, A. Jentzen, and W. E, *Solving high-dimensional partial differential equations using deep learning*, Proc. Natl. Acad. Sci. USA, 115 (2018), pp. 8505–8510, https://doi.org/10.1073/pnas.1718942115.
[19] C. Huré, H. Pham, A. Bachouch, and N. Langrené, *Deep neural networks algorithms for stochastic control problems on finite horizon, part I: convergence analysis*, 2018, https://arxiv.org/abs/1812.04300.
[20] D. Izzo, E. Öztürk, and M. Märtens, *Interplanetary transfers via deep representations of the optimal policy and/or of the value function*, in Genetic and Evolutionary Computation Conference, 2019, pp. 1971—1979, https://doi.org/10.1145/3319619.3326834.
[21] F. Jiang, G. Chou, M. Chen, and C. J. Tomlin, *Using neural networks to compute approximate and guaranteed feasible Hamilton-Jacobi-Bellman PDE solutions*, 2016, https://arxiv.org/abs/1611.03158.
[22] D. Kalise and K. Kunisch, *Polynomial approximation of high-dimensional Hamilton-Jacobi-Bellman equations and applications to feedback control of semilinear parabolic PDEs*, SIAM J. Sci. Comput., 40 (2018), pp. A629–A652, https://doi.org/10.1137/17M1116635.
[23] W. Kang, P. De, and A. Isidori, *Flight control in a windshear via nonlinear h_∞ methods*, in Proceedings of the 31st IEEE Conference on Decision and Control, vol. 1, 1992, pp. 1135–1142.

[24] W. Kang and L. C. Wilcox, *A causality free computational method for HJB equations with application to rigid body satellites*, in AIAA Guidance, Navigations, and Control Conference, 2015, pp. 1–10, https://doi.org/10.2514/6.2015-2009.
[25] W. Kang and L. C. Wilcox, *Mitigating the curse of dimensionality: Sparse grid characteristics method for optimal feedback control and HJB equations*, Comput. Optim. Appl., 68 (2017), pp. 289–315, https://doi.org/10.1007/s10589-017-9910-0.
[26] J. Kierzenka and L. F. Shampine, *A BVP solver based on residual control and the MATLAB PSE*, ACM Trans. Math. Softw., 27 (2001), pp. 299–316, https://doi.org/10.1145/502800.502801.
[27] D. Liberzon, *Calculus of Variations and Optimal Control Theory: A Concise Introduction*, Princeton University Press, Princeton, NJ, 2011, https://doi.org/10.2307/j.ctvcm4g0s.
[28] D. Lukes, *Optimal regulation of nonlinear dynamical systems*, SIAM J. Control, 7 (1969), pp. 75–100, https://doi.org/10.1137/0307007.
[29] O. L. Mangasarian, *Sufficient conditions for the optimal control of nonlinear systems*, SIAM J. Control, 4 (1966), pp. 139–152, https://doi.org/10.1137/0304013.
[30] T. Nakamura-Zimmerer, Q. Gong, and W. Kang, *A causality-free neural network method for high-dimensional Hamilton-Jacobi-Bellman equations*, in American Control Conference (ACC), 2020, pp. 787–793, https://doi.org/10.23919/ACC45564.2020.9147270.
[31] C. Navasca and A. J. Krener, *Patchy Solutions of Hamilton-Jacobi-Bellman Partial Differential Equations*, Springer, Berlin-Heidelberg, 2007, pp. 251–270, https://doi.org/10.1007/978-3-540-73570-0_20.
[32] S. Osher and J. A. Sethian, *Fronts propagating with curvature-dependent speed: Algorithms based on Hamilton-Jacobi formulations*, J. Comput. Phys., 79 (1988), pp. 12–49, https://doi.org/10.1016/0021-9991(88)90002-2.
[33] M. Raissi, P. Perdikaris, and G. Karniadakis, *Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations*, J. Comput. Phys., 378 (2019), pp. 686–707, https://doi.org/10.1016/j.jcp.2018.10.045.
[34] J. Sirignano and K. Spiliopoulos, *DGM: A deep learning algorithm for solving partial differential equations*, J. Comput. Phys., 375 (2018), pp. 1339–1364, https://doi.org/10.1016/j.jcp.2018.08.029.
[35] Y. Tassa and T. Erez, *Least squares solutions of the HJB equation with neural network value-function approximators*, IEEE Trans. Neural Netw., 18 (2007), pp. 1031–1041, https://doi.org/10.1109/TNN.2007.899249.
[36] L. N. Trefethen, *Spectral Methods in MATLAB*, Society for Industrial and Applied Mathematics, Philadelphia, PA, 2000, https://doi.org/10.1137/1.9780898719598.
[37] P. Virtanen, R. Gommers, T. E. Oliphant, and et. al., *SciPy 1.0: Fundamental algorithms for scientific computing in Python*, Nat. Methods, 17 (2020), pp. 261–272, https://doi.org/10.1038/s41592-019-0686-2.
[38] I. Yegorov and P. M. Dower, *Perspectives on characteristics based curse-of-dimensionality-free numerical approaches for solving Hamilton-Jacobi equations*, Appl. Math. Optim., (2018), https://doi.org/10.1007/s00245-018-9509-6.