Learning Solutions to Partial Differential Equations

Arasu Arun (aa7977), Nitin Shyamkumar (nhs9171), Chia-Hao (Leo) Wu (chw398)

December 15, 2020

Abstract

Physics informed machine learning consists of various machine learning methods for learning partial differential equation (PDE) models and solutions. We empirically study three methods from this field for learning solutions to a PDE model, focusing on the Hamilton Jacobi Bellman PDE for continuous time control. We find that the methods learn accurate solutions for high dimensional linear systems. Although the methods have some shortcomings, they achieve promsing results on the Cartpole task from nonlinear control.

1 Introduction

Deep learning combined with recent surges in compute power have led to transformative discoveries in a wide range of disciplines. Advancements in computer vision [1, 2], reinforcement learning methods for playing complex games [3], and most recently problems in structural biology [4] encouraged applied mathematicians and physicists to use machine learning (ML) methods and define a new area of physics informed machine learning. These approaches propose deep networks, among other ML methods, to solve challenging nonlinear problems in the physical sciences [5] focusing on both solving Partial Differential Equations (PDEs) [6] as well as systems identification (learning parameters of a model) [7]. In these approaches, domain experts can structure the machine learning approach to include well established physical invariants. The approach has a few benefits. Firstly, a physical system can be highly nonlinear and unconstrained optimization may be difficult, so establishing physical constraints can reduce the search space of ML methods. Secondly, physical constraints increase trust in the model by ensuring that the learned model respects well established theoretical or empirical laws of nature.

Separately, computer science researchers in reinforcement learning (RL) have continued researching planning algorithms for Markov Decision Processes. In this framework, the challenge is to plan an optimal policy in known environment with established dynamics. While this problem has received substantial attention from the RL community, comparatively less attention has been directed at combining traditional control theoretic frameworks with advancements in deep learning. Many recent methods, such as the Deep Q Network, have moved in entirely the other direction, abstracting the physical system away entirely from the learning problem. Nonetheless, researchers in the machine learning community have continued to explore the intersection of traditional mathematical control theory and reinforcement learning. [8] proposes a method to identify (learn, in ML parlance) a nonlinear system under the broad assumption that the function f describing system dynamics lies in the Reproducing Kernel Hilbert Space (RKHS) of a known kernel. A similar approach is taken by [9].

In our work, we examine the effectiveness of physics informed machine learning methods for the planning task in reinforcement learning (RL). We propose making stronger assumptions than [8], assumptions that are perhaps stronger than typical in computer science RL literature. As [9] observes, such an assumption is not necessarily unreasonable. In applications such as robotics, we may have accurate estimates of internal system components including mass and dimensions. We study this approach in detail, comparing two different physics informed neural network methods with a Deep Q Network baseline and discuss a few theoretical aspects. We finally study the methods on a simple Linear Quadratic Gaussian problem to establish a base competency, as well as the canonical challenging and highly nonlinear cart pole system.

2 Hamilton Jacobi Bellman Partial Differential Equations

2.1 Stochastic Optimal Control

We present the technical formulation of the stochastic optimal control problem, discussing only relevant highlights. For additional details, see the appendix. As discussed in class, the discrete time, finite horizon planning problem for a Markov Decision Problem with known system dynamics is the problem of finding the optimal policy π^* and corresponding value function for each state given by

$$s_t = \delta(s_{t-1}, \pi^*(s_{t-1}))$$
$$V_{\pi^*}(s) = \max_{\pi} \mathbb{E}\left[\sum_{t=0}^{T-1} r(s_t, \pi^*(s_t)) \;\middle|\; s_0 = s\right]$$

In this problem, $s \in S$ is a state in the system, (s_t) is the state sequence, δ is the transition function, which may be probabilistic in nature, $r(s,a)$ is the reward function for a state-action pair, T is the horizon, π is the policy, and V is the value function [10]. The continuous time formulation bears a strong resemblance to the problem above.

The standard stochastic continuous time control problem is described as follows. A full explanation is outside the scope of this report, we direct the reader to [11] and [12]. Here, we have a state trajectory $\mathbf{x}(t) \in \mathbb{X} \subseteq \mathbb{R}^n$, a control trajectory $\mathbf{u}(t) \in \mathcal{U} \subset \mathbb{R}^c$, a rolling cost $q : \mathbb{R}^n \times \mathbb{R}^c \to \mathbb{R}$, and a terminal cost $q_T : \mathbb{X} \to \mathbb{R}$. The dimensions n and c are the state and input control dimensions respectively. $\mathbf{f}$ specifies the system dynamics. W_t is the standard Wiener process (Brownian motion). The problem is to find the optimal control u^* (usually u^* is restricted in some way; we restrict $u^* : [0,T] \to \mathcal{U}$ to the standard $\mathcal{L}^2$ space) that minimizes the cost function J.

$$d\mathbf{x} = \mathbf{f}(\mathbf{x}(t), \mathbf{u}^*(t))dt + G(t, \mathbf{x}(t), \mathbf{u}^*(t))dW_t \tag{1}$$
$$J^*(t, \mathbf{x}(t)) = \min_{\mathbf{u}^*} \left\{ q_T(\mathbf{x}(T)) + \int_{t=0}^{T} q(\mathbf{x}(t), \mathbf{u}^*(t))\, dt \right\} \tag{2}$$

This is analgous to the discrete time formulation, with the exception that continuous time control literature largely follows the convention of minimizing cost instead of maximizing reward. We follow this convention for the rest of the report.

The reader may note the seeming inconsistency of claiming to know the system dynamics, but including stochasticity (W_t) nonetheless. First, it is a common approach for modeling real world dynamical systems where parameter estimates may have uncertainty. Alternatively, stochasticity can be added as a mechanism for handling complex nonlinear effects that the model does not account for. Finally, stochasticity can account for noise in the control inputs of the system.

The control problem in equations (1) and (2) leads to the well known Hamilton Jacobi Bellman PDE (and boundary). If V satisfies this equation, then $V = J^*$ (the optimal cost to go) [12]. The derivation is again, outside the scope of this report, and can be seen in full detail in [12, p. 182].

$$\frac{\partial V}{\partial t} = \min_{\mathbf{u}(t)} \left\{ q(\mathbf{x}, \mathbf{u}) + \langle \nabla_{\mathbf{x}} V, f(\mathbf{x}, \mathbf{u}) \rangle + \frac{1}{2} Tr[Hess_{\mathbf{x}} V G(t, \mathbf{x}, \mathbf{u}) G(t, \mathbf{x}, \mathbf{u})^T] \right\}$$
$$V(T, \mathbf{x}) = q_T(\mathbf{x}(T))$$

One important observation is that the stochasticity disappears in this formulation, appearing only as the product of the Hessian and the noise covariance matrix in the last term. This can intuitively be understood as an additional diffusive term; when $G = \sigma I$, the term becomes a scalar multiple of the Laplacian ΔV. Solving for V in the general case is highly nontrivial. We restrict our attention to a special class of functions f where f is semilinear with respect to $\mathbf{u}$, or $f(\mathbf{x}, \mathbf{u}) = a(\mathbf{x}) + B(\mathbf{x})\mathbf{u}$. Furthermore, we also limit our analysis

to constant quadratic cost. That is, $q(\mathbf{x}, \mathbf{u}) = \frac{1}{2}(\mathbf{x}^T Q \mathbf{x} + \mathbf{u}^T R \mathbf{u})$ while $q_T(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T Q_T \mathbf{x}$. In this setting, we find that the optimal control $\mathbf{u}^* = -R^{-1} B(\mathbf{x})^T \nabla_{\mathbf{x}} V$. Unsurprisingly, this is a policy which moves in the direction of decreasing cost. Furthermore, we also restrict $G(t, \mathbf{x}, \mathbf{u})$ to be a multiple of the identity matrix, that is, $G = \sigma I$, where σ controls the noise. These assumptions result in the following HJB PDE:

$$\frac{\partial V}{\partial t} = \frac{1}{2}\mathbf{x}^T Q \mathbf{x} - \frac{1}{2}(\nabla_{\mathbf{x}} V)^T B(\mathbf{x}) R^{-1} B(\mathbf{x})^T (\nabla_{\mathbf{x}} V) + \langle \nabla_{\mathbf{x}} V, a(\mathbf{x}) \rangle + \frac{\sigma^2}{2} Tr[Hess_{\mathbf{x}} V] \tag{3}$$

$$V(T, \mathbf{x}) = \frac{1}{2}\mathbf{x}(T)^T Q_T \mathbf{x}(T) \tag{4}$$

2.2 Semilinear Parabolic Partial Differential Equations

Under the assumptions specified in the previous section for the specified stochastic control task, particularly the restriction of $f(\mathbf{x}, \mathbf{u})$ to the space of semilinear functions $f(\mathbf{x}, \mathbf{u}) = a(\mathbf{x}) + B(\mathbf{x})\mathbf{u}$, the resulting Hamilton Jacobi Bellman (HJB) PDE takes on a form known as a semilinear parabolic PDE. A semilinear parabolic PDE has the following form:

$$\frac{\partial v}{\partial t} + \varphi(t, x, v, \nabla_x v) + \mu(t, x, v, \nabla_x v)^T \nabla_x v(t, x) + \frac{1}{2} Tr\left[Hess_{\mathbf{x}} v(t, x) G(t, x, v) G(t, x, v)^T\right] = 0 \tag{5}$$

d2q‘ where v is the function of interest, of both t and $\mathbf{x}$, φ is a general nonlinear function, and μ is a vector valued function of $t, x, v, \nabla_x v$. While one of the methods we discuss can use this form of the HJB PDE directly, two of the other methods we discuss use a mathematical trick instead to enhance the tractability of learning the function v. [13] showed that the solution to equation (5) is also a solution to the following Forwards-Backwards Stochastic Differential Equation (SDE) [1]

$$dX_t = \mu(t, X_t, Y_t, Z_t)\, dt + \sigma(t, X_t, Y_t)\, dW_t, \quad t \in [0, T] \tag{6}$$

$$X_0 = \xi \tag{7}$$

$$dY_t = \varphi(t, X_t, Y_t, Z_t)\, dt + Z_t' \sigma(t, X_t, Y_t)\, dW_t, \quad t \in [0, T) \tag{8}$$

$$Y_T = g(X_T) \tag{9}$$

where ξ is the initial state and $g(X_T)$ is the boundary condition. In this framework, solving the HJB PDE for a stochastic control problem can be understood as solving for a controlled diffusion over the relevant domain.

3 Methods

3.1 Deep Backwards Stochastic Differential Equation Network

In [14], Han et. al develop the Deep Backwards Stochastic Differential Equation Network (Deep BSDE Network) that utilizes the stochastic differential equation formulation of equation 5 to approximate a solution to the HJB PDE of interest. The key difference from equation 5 is that μ and σ are restricted so that $\mu(t, X_t, Y_t, Z_t) = \mu(t, X_t)$ and similarly $\sigma(t, X_t, Y_t) = \sigma(t, X_t)$.

[1] The more canonical form in most literature concerning SDEs is the following, where the integral is understood to be the standard Itô integral in stochastic calculus.

$$X_t = \xi + \int_0^t \mu(s, X_s, Y_s, Z_s)\, ds + \int_0^t \sigma(s, X_s, Y_s)\, dW_s$$

$$Y_t = g(X_T) + \int_t^T \varphi(s, X_s, Y_s, Z_s)\, ds + \int_t^T Z_t' \sigma(s, X_s, Y_s)\, dW_s$$

However, the deep neural network methods that we present are best understood as using the differential form to learn a particular discretization scheme, so we limit our attention to the selected form.

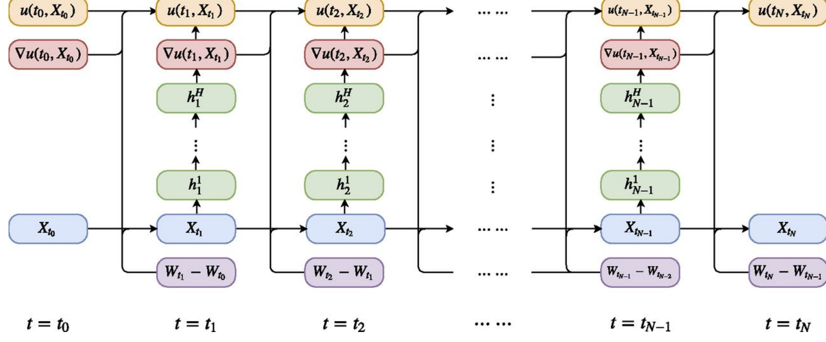


Figure 1: The Deep BDSE architecture, from [14]

Architecture

Figure 1 illustrates the system architecture. Given the equivalent SDE outlined in equation 6 to the parabolic PDE described by equation 5, with specified boundary conditions, we construct a neural network architecture as follows. Select a hyper parameter N to divide our domain T into N discretization steps, each of size $dt = T/N$. Observe that $W_{t_i} - W_{t_{i-1}} \sim N(0, dt)$. We will construct a network that learns the gradient of v (listed as u in 1) at each discretization step, and then forward propagates a trajectory using this gradient estimate. Finally, the end state of the trajectory at time T is compared to the terminal cost function, with an appropriate loss function applied to the difference. More specifically, $\hat{v}$ is the neural network approximation for v and is defined by the following three connections:

1. Given a neural network with parameters θ, compute $\nabla \hat{v}(t_n, X_{t_n}) = f_\theta(X_{t_n})$
2. Compute $\hat{v}(t_{n+1}, X_{t_{n+1}}) = \hat{v}(t_n, X_{t_n}) - f\left(t_n, X_{t_n}, \hat{v}(t_n, X_{t_n}), \nabla \hat{v}(t_n, X_{t_n})\right) dt + \nabla \hat{v}(t_n, X_{t_n})^T \sigma(t_n, X_{t_n})(W_{t_{n+1}} - W_{t_n})$
3. Compute $X_{t_{n+1}} = X_{t_n} + \mu(t_n, X_{t_n})dt + \sigma(t_n, X_{t_n})(W_{t_{n+1}} - W_{t_n})$

Loss Function

We then optimize the network parameters θ with the architecture and paths X_{t_n} specified above using the following loss function:

$$\mathcal{L}(\theta) = \mathbb{E}\left[|g(X_{t_n}) - \hat{v}(X_{t_n}, W_{t_n})|\right]$$

Training

Training is done by randomly sampling to construct Wiener process samples (W_t) for all t, propagating X_t using μ and σ (as specified in the connection (3)), and then applying the neural network approximation for $\hat{v}$. The Loss function $\mathcal{L}(\theta)$ is minimized using the Adam optimizer.

3.2 Forward Backward Stochastic Neural Network

Building off the Deep BSDE method from [14], [15] considers the original stochastic differential equation presented in 6 without restricting μ or σ.

Architecture

The chosen SDE model leads to what is known as the Euler-Maruyama discretization scheme for SDEs, given by:

$$\begin{aligned} X_{n+1} &\approx X_n + \mu\left(t_n, X_n, Y_n, Z_n\right) \Delta t_n + \sigma\left(t_n, X_n, Y_n\right) \Delta W_n \\ Y_{n+1} &\approx Y_n + \varphi\left(t_n, X_n, Y_n, Z_n\right) \Delta t_n + \left(Z_n\right)' \sigma\left(t_n, X_n, Y_n\right) \Delta W_n \end{aligned}$$

This leads to an architecture to approximate the solution v as $\hat{v}$ with a neural network parameterized by θ. As before, we select N, the size of our time discretization mesh, and divide T/N to get dt. We construct two types of connections, but use a single neural network to approximate $\hat{v}$. In what follows, $\hat{v}$ is understood to be parameterized by θ ($\hat{v}_\theta$):

- Compute $X_{t_{n+1}} = \mu\left(t_n, X_{t_n}, \hat{v}(t_n, X_{t_n}), \nabla_{X_{t_n}} \hat{v}(t_n, X_{t_n})\right) \cdot dt + \sigma\left(t_n, X_n, \hat{v}(t_n, X_{t_n})\right) dW_{t_n}$
- Define $\tilde{Y}_{t_{n+1}} = \hat{v}(t_n, X_{t_n}) + \varphi(t_n, X_{t_n}, \hat{v}(t_n, X_{t_n}), \nabla_{X_{t_n}} \hat{v}(t_n, X_{t_n}))dt + \left(\nabla_{X_{t_n}} \hat{v}(t_n, X_{t_n})\right)^T \sigma\left(t_n, X_{t_n}, \hat{v}(t_n, X_{t_n})\right) dW_{t_n}$

Loss Function

Using the specified architecture, we seek to minimize the following loss with respect to θ (recall $\hat{v}$ is actually $\hat{v}_\theta$). Define the loss function as

$$\mathcal{L}(\theta) = \mathbb{E}_{W_t}\left[(Y_{t_N} - g(X_{t_N}))^2 + \sum_{i=0}^{N-1}\left(Y_{t_{n+1}} - \hat{v}_\theta\left(t_{n+1}, X_{t_{n+1}}\right)\right)^2\right]$$

Training

Randomly sample a Wiener process W_t and use this to simulate a trajectory through space and time with a specified initial condition as indicated above. Use the Adam optimizer to minimize loss.

3.3 Deep Galerkin Method

Sirignano et. al proposed the Deep Galerkin Method in [16]. Consider a parabolic PDE with d spatial dimensions:

$$\begin{aligned} \partial_t u(t,x) + \mathcal{L}u(t,x) &= 0 \qquad && (t,x) \in [0,T] \times \Omega \\ u(t=0,x) &= u_0(x) && x \in \Omega \\ u(t,x) &= g(t,x) && x \in [0,T] \times \partial\Omega \end{aligned}$$

where $\partial\Omega$ is the boundary of the domain and $x \in \Omega \subset \mathbb{R}^d$. This method uses stochastic gradient descent at randomly sampled points in space to minimize a three part loss objective composed of loss functions on the differential operator, the initial condition, and the boundary conditions [2]. We describe this in more detail.

Architecture

The architecture used in the DGM method is a model composed of LSTM type layers. Each layer is assumed to have M parameters with a nonlinear activation function ϕ. Each layer contains "sub-layers" of computations. The important feature is the repeated element-wise multiplication of nonlinear functions of the input. This helps to model more complicated functions which are rapidly changing in certain time and space regions. The key hyperparameters in the neural network are the number of layers L, the number of units M in each sub-layer, and the choice of activation unit $\phi(y)$. Given the architecture's complexity, we give the full specification in the appendix.

[2] [16] proved a key theorem, that a neural network approximator is also an approximation to the solution of the PDE. The theorem is sketched as follows. Define $\mathfrak{C}^n$ as the class of neural networks with n hidden units and let f^n with n hidden units which minimizes $J(f)$.

$$\begin{aligned} &\text{there exists } f^n \in \mathfrak{C}^n \text{ such that } J(f^n) \to 0, \text{ as } n \to \infty, \text{ and} \\ &\qquad\qquad\qquad\qquad\qquad\qquad f^n \to u \text{ as } n \to \infty \end{aligned}$$

Then, one can learn the function f which minimizes $J(f)$ using stochastic gradient descent on a sequence of time and space points drawn at random from Ω and $\partial\Omega$, and the function f approximates the PDE solution $u(t,x)$. Unfortunately, this theorem does not specify a rate of convergence which means it is of limited benefit to our problem.

Loss Function

The goal is to find a set of parameters θ such that the function $f(t, x; \theta)$ minimizes the error $J(f)$. If the error $J(f)$ is small, then $f(t, x; \theta)$ will closely satisfy the PDE differential operator, boundary conditions, and initial condition. Then, the loss function is:

$$J(f) = \|\partial_t f(t, x; \theta) + \mathcal{L} f(t, x; \theta)\|^2_{2,[0,T]\times\Omega} + \|f(t, x; \theta) - g(t, x)\|^2_{2,[0,T]\times\partial\Omega} + \|f(0, x; \theta) - u_0(x)\|^2_{2,\Omega}$$

Training

The DGM algorithm is:
1. Generate random points (t_n, x_n) from $[0, T] \times \Omega$ and (τ_n, z_n) from $[0, T] \times \partial\Omega$ according to respective probability densities v_1 and v_2. Also, draw the random point w_n from Ω with probability density v_3.
2. Calculate the squared error $G(\theta_n, s_n)$ at the randomly sampled points $s_n = \{(t_n, x_n), (\tau_n, z_n), w_n\}$ where:

$$G(\theta_n, s_n) = (\frac{\partial f}{\partial t}(t_n, x_n; \theta_n) + \mathcal{L} f(t_n, x_n; \theta_n))^2 + (f(\tau_n, z_n; \theta_n) - g(\tau_n, z_n))^2 + (f(0, w_n; \theta_n) - u_0(w_n))^2$$

3. Take a descent step at the random point s_n:

$$\theta_{n+1} = \theta_n - \alpha_n \nabla_\theta G(\theta_n, s_n)$$

4. Repeat until convergence criterion is satisfied.

3.4 Deep Q-Network

Mnih et. al proposed the deep Q-network(DQN) approach in [17] that combines a modified version of online Q-learning with deep neural networks. We use this model as a baseline. This approach trains the model to choose the best action for given observations and rewards on the terminal states.

Architecture

The input to the neural network consists of a sequence of preprocessed data produced by the preprocessing map ϕ from a sequence of sample states. The first hidden layer convolves filters and applies a rectifier nonlinearity. The second hidden layer convolves filters again followed by a rectifier nonlinearity. This is followed by a third convolutional layer that convolves filters followed by a rectifier. The final hidden layer is fully-connected and consists of rectifier units. The output layer is a fully-connected linear layer with a single output for each valid action. See Figure 2

The goal is to train a model to be able to select actions to maximize future rewards. We can define the optimal action-value function $Q^*(s, a)$ as the maximum expected return achievable by following any policy, after seeing some sequence s and then taking some action a. Then, we estimate the action-value function by using the Bellman equation as an iterative update, $Q_{i+1} = E_{s'}[r + \gamma \max_{a'} Q_i(s', a')|s, a]$ where s' stands for the next time-step state sequence and a' stands for all possible actions. Such value iteration algorithms converge to the optimal action-value function, $Q_i \to Q^*$ as $i \to \infty$. In practice, we refer to a neural network function approximator with weights θ as a Q-network. Then, it trains the Q-network by adjusting the parameters θ_i at iteration i to reduce the mean-squared error in the Bellman equation.

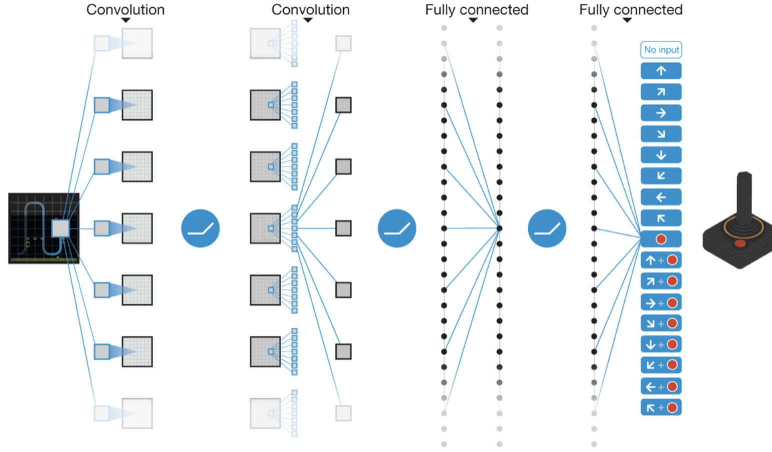
Loss Function

It then leads to a sequence of loss functions $L_i(\theta_i)$ that changes at each iteration i,

$$\begin{aligned} L_i(\theta_i) &= E_{s,a,r}[(E_{s'}[y|s, q] - Q(s, a; \theta_i))^2] \\ &= E_{s,a,r,s'}[(y - Q(s, a; \theta_i))^2] + E_{s,a,r}[V_{s'}[y]] \end{aligned}$$

where y stands for approximate target values, $y = r + \gamma \max_{a'} Q(s', a'; \theta_i^-)$ and θ_i^- stands for the weight parameter θ from previous iteration. Then, it optimizes the loss function by stochastic gradient descent.

Schematic illustration of the convolutional neural network.



V Mnih *et al.* *Nature* **518**, 529-533 (2015) doi:10.1038/nature14236

nature

Figure 2: The DQN architecture, from [17]

Method	Average Error	Training Time (s)
Deep BSDE	0.0018	118
FBSNN	0.00214	83
DGM	0.00180	242

Table 1: Results from running the methods over 3 trials.

Training

It draws samples (or minibatches) of experience uniformly at random from the pool of stored samples and applies Q-learning, updating the weight parameters using stochastic gradient descent.

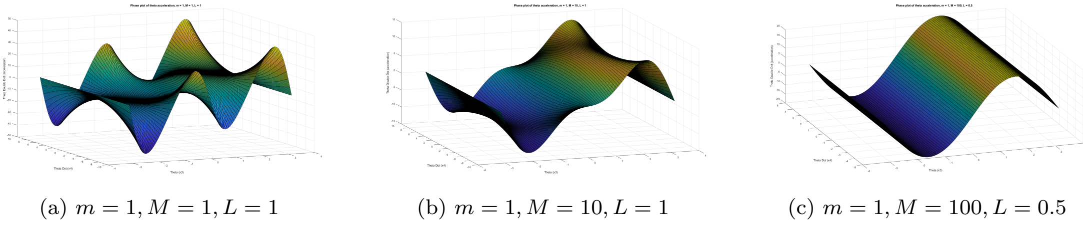
4 Experiments

4.1 Linear Quadratic Gaussian

We first evaluate a Linear Quadratic Gaussian task with a known semi-analytical solution to test the efficacy of various methods and their accuracy. We consider a problem where $Q = R = I$, and $Q_T(\mathbf{x}) = \ln\left(\frac{1+||\mathbf{x}||^2}{2}\right)$. $\sigma = \sqrt{2}I$, while $f(\mathbf{x}) = 2\sqrt{\lambda}\mathbf{u}$. To test the efficacy of this method in learning high dimensional value functions, we consider a 100 dimensional case. The HJB PDE for this control task is

$$\frac{\partial u}{\partial t} = \lambda||u(\mathbf{x},t)||^2 - \Delta u(\mathbf{x},t)$$

with a boundary condition specified by Q_T. It admits the solution $u(t,\mathbf{x}) = -\frac{1}{\lambda}\ln\left(\mathbb{E}\left[\exp\left(-\lambda g\left(\mathbf{x} + \sqrt{2}W_{T-t}\right)\right)\right]\right)$ [14]. We use this solution to compute the accuracy of our method, and discuss its evaluation in more detail in the appendix. Note that the optimal control for this solution is to proceed in the direction of the origin with an appropriate normalization constant. Results are listed in Table 1.



(a) $m = 1, M = 1, L = 1$ (b) $m = 1, M = 10, L = 1$ (c) $m = 1, M = 100, L = 0.5$

Figure 3: Nonlinear dynamics of the cartpole system. Phase plots for $\ddot{\theta}$ as a function of θ and $\dot{\theta}$, $x, \dot{x}$ are both fixed to 0.

4.2 Cart-Pole

Setup

The Cart-Pole (CP) problem is a standard benchmark in RL. The objective is to keep the inverted pendulum linked to a cart in an upright position, using only movements to the left or to the right. We model the dynamical system fully using the Newtonian mechanics derivation from [18]. The system parameters are M, the mass of the cart; m, the mass of the pendulum; L, the pendulum length; x, the cart's position on the x-axis; θ, the angle of the pendulum from vertical; and finally $u(t)$, the control to the cart. We obtain the following dynamical system with state space $\mathbf{x} = [x_1, x_2, x_3, x_4]^T = [x, \dot{x}, \theta, \dot{\theta}]^T$

$$\dot{\mathbf{x}} = \mathbf{a}(\mathbf{x}) + B(\mathbf{x})\mathbf{u}(t) \tag{10}$$

$$r := \frac{m}{M+m} \tag{11}$$

$$\mathbf{a}(\mathbf{x}) = \begin{bmatrix} x_2 \\ rLx_4^2 \sin x_3 - \frac{rL\cos x_3}{L\left(\frac{4}{3} - r\cos^2 x_3\right)}\left(g \sin x_3 - 0.5Lrx_4^2 \sin 2x_3\right) \\ x_4 \\ \frac{1}{L\left(\frac{4}{3} - r\cos^2 x_3\right)}\left(g \sin x_3 - 0.5Lrx_4^2 \sin 2x_3\right) \end{bmatrix} \tag{12}$$

$$B(\mathbf{x}) = \begin{bmatrix} 0 \\ \frac{r^2 \cos^2 x_3}{m\left(\frac{4}{3} - r\cos^2 x_3\right)} + \frac{r}{m} \\ 0 \\ \frac{r}{mL\left(\frac{4}{3} - r\cos^2 x_3\right)} \cos x_3 \end{bmatrix} \tag{13}$$

These equations, derived in [18], describe the system fully, but are highly nonlinear. A few phaseplots for different parameter combinations in Figure 3 illustrate the extent of the nonlinearities for different parameter combinations. For the OpenAI Gym example we consider, they are quite strong.

In practice, a common approach to make this system tractable is to linearize the dynamics model around the point $\vec{0}$, resulting in the well known Linear Quadratic Regulator (LQR) setup, which is described and solved in [19]. However, linearization involves restrictive assumptions about the physical system. [19] makes three assumptions: small angular velocity ($\dot{\theta} \ll 1$), $m \ll M$ so $m/M \approx 0$, and that sin and cos can be approximated with the small angle assumption, so $\sin\theta \approx \theta$ and $\cos\theta \approx 1$. These assumptions are only valid over a very narrow range of the state space where θ is in the range $\pm 8°$. [3] Previous attempts at nonlinear control have generally opted to solve a related but slightly different formulation of this problem by using a log transform to yield a PDE for a "desirability" function, rather than the value function [20]. Many of these approaches use traditional grid methods. These also place a barrier on translating results back to the

[3] This is an assumption worth discussing in more detail. In the OpenAI Gym Cartpole environment, as with many other formulations of the problem, the cartpole starts with the pendulum at the top. It's completely reasonable that an LQR controller in this regime would actually do quite well, since it may be the case that θ doesn't exceed the narrow range of $\pm 8°$. However, our aim was to show that we could learn a far more general strategy that would work even if the system was initialized in difficult conditions, such as with the pendulum at the bottom.

heightMethod	Reward Reached (max 200)	Training Time Required
FBSNN	200	$\sim$ 20 seconds
DQN	200	$\sim$ 80 seconds
Simple	40	-

Table 2: Results from Cartpole simulation, averaged over 10 attempts.

original problem, as well as recovering the optimal policy. Our proposed use of neural networks thus suggests a way to solve the problem entirely while avoiding these shortcomings.

Results

We run the FBSNN method on the cartpole problem from the OpenAI Gym Cartpole-v1 environment, which has parameters $m = 0.1$, $M = 1$, and $L = 1$. In our discussion section, we discuss why we did not apply the BDSE method and the DGM. Using the learned value function V, we can identify the optimal control action at any point in space by calculating $\mathbf{u}^* = -R^{-1}B(\mathbf{x})^T\nabla_{\mathbf{x}}V$. We then apply simple thresholding to recover the control. We began our training by fixing $Q = Q_T = R = I$, $G = \sigma I$, $\sigma = 0.01$, $T = 10s$, although these were subject to change. We apply these neural network and control strategies to the OpenAI Gym Cartpole-v1 environment, and report their average episode reward in Table 2. The "Simple" strategy simply chooses the direction that the pendulum is currently leaning.

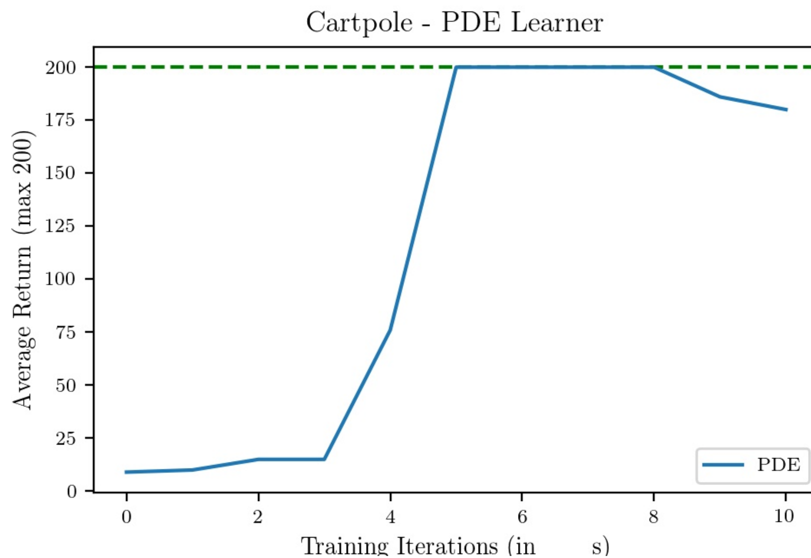
Figures 4a and 4b show the training curves of the FBSNN PDE learner and the DQN method for the Cartpole problem. We measured the performance of the methods at intervals over their training time, until they reached their peak. Performance is measured in the number of steps the cartpole stayed up (averaged over 10 attempts).

Discussion

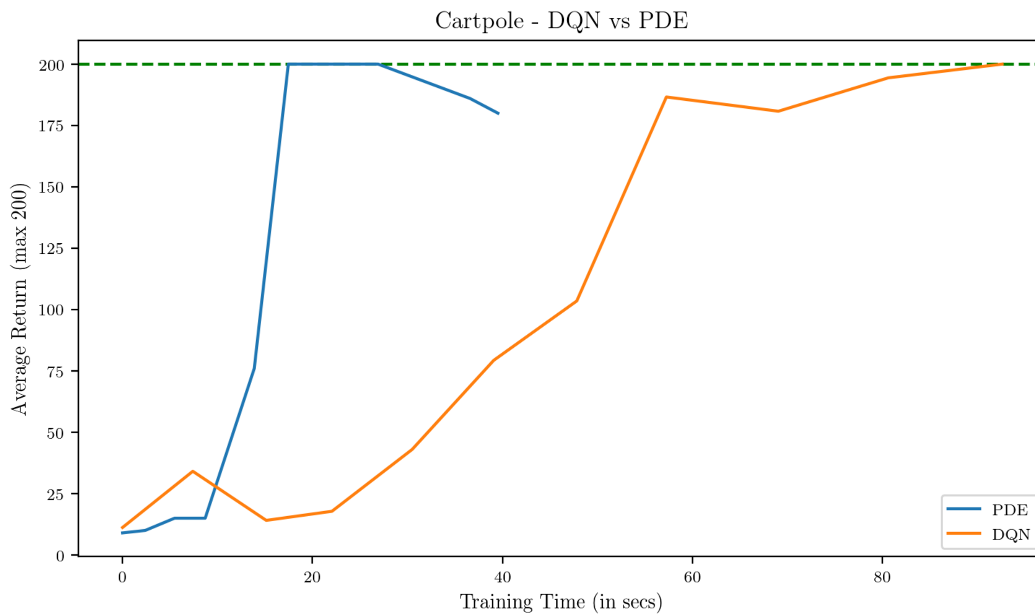
Although we were able to achieve our goals of learning a solution to the desired PDE for both examples, the results were mixed. All methods performed fairly well in learning an approximate solution to the LQG, as seen in Table 1. But limitations in the Deep BSDE method and a poor showing by the DGM meant that the FBSNN was the only method that performed reasonably on the Cartpole example. We analyze the results in more detail below.

The Deep BSDE method was architecturally limited for longer time horizons (T) critical to the control problem. The Deep BSDE method worked well in the LQG case because the time horizon was short. But because the number of parameters scales with the size of the discretization mesh, for larger time horizons such as in the Cartpole problem ($T = 4$), this method soon became intractable, especially when we wanted to experiment with even longer time horizons. The problem is not solved by reducing the discretization mesh size, since this then means that the learned solution becomes less accurate. [5] also pointed out that the Deep BSDE method cannot be used to infer values outside of the discretization mesh since each discrete timstep is a layer $[0, T]$.

The Deep Galerkin Method struggled to learn the full nonlinear system without additional constraints. While the Deep Galerkin Method performed very well in the LQG setting, it struggled to cope with the highly nonlinear HJB PDE found in the Cartpole problem. The results were too poor to use. We leave some notes quantifying this failure in the appendix. Unlike the SDE based methods, which likely benefitted from decoupling components of the HJB PDE into two separate stochastic trajectories, the Deep Galerkin Method had no such constraints narrowing the search space. As noted earlier, we sought to learn the value function, not the often learned "desirability" function resulting from a log transform of the HJB PDE, as suggested in [20]. Since the proponents of the DGM suggested that it could achieve promising results on nonlinear PDEs, we opted to test our network on the untransformed PDE. A possible avenue for work would be to test the DGM on these transformed PDEs, though its unclear what benefits such an approach would yield over the conventional methods designed for this transformed desirability PDE.



(a) The return (number of steps the cartpole stays balanced) of the FBSNN (PDE) Learner. The maximum return of 200 was reached after around 5500 iterations. Beyond 8000 iterations of training, the performance decreases.



(b) Comparison of the training plots of the FBSNN (PDE) learner and DQN. The PDE learner is able to learn the solution much faster than the DQN. On the other hand, we find that the DQN learner is more robust and its performance does not decrease with more training steps.

The FBSNN method performed well, consistently beating Deep Q-Learning by reaching an optimal solution faster. Figure 4b illustrates this pattern, which was observed consistently over multiple trajectory runs (see Figure 4a as well). On average, the training time for the FBSNN method was 25% of the required training time for Deep Q-Learning.

The FBSNN method seem to be highly variable in recovering a solution. While we generally recovered an optimal solution, exactly when we were able to do so seemed unpredictable. The FBSNN's benefit of generalizing to times different from the discretization mesh seems to come at the expense of higher variability. On one hand, we nearly always recovered an optimal (200 reward) solution with sufficient training time. But on the other, additional training didn't always help and sometimes degraded performance, see 4b. One possibility was overparameterization of our networks, but a hyperparameter scan of layer sizes didn't yield effective improvements (see notes in Appendix with quantitative characteristics). We believe that the highly stochastic training paradigm suggests that the simulated trajectories may impart substantial variance in the trained model. These results suggest that future variants of these methods will need to consider regularizing and the neural network's vulnerability to PDE misspecification.

On the other hand, another interesting, and potentially conflicting observation comes from an accidental mistake made earlier in our experiments. One of the authors made a mistake in an initial attempt to derive equation 10 (assuming a point mass for the pendulum which leads to a different moment of inertia) leading to a few constant factors differing slightly in the derived system. Despite these errors, the value function and ensuring control still led to optimal policies that achieved perfect 200 reward. Such a mistake contradicts our presumption earlier that our network was overfitting although it is possible that the author's mistake had sufficiently low impact.

5 Conclusion

All methods showed some promise empirically on a simplified LQG example. But structural limitations for the Deep BSDE method and the complexity of learning a full model without constraints for the Deep Galerkin Method proved to be their downfall. The FBSNNs performed well on the Cartpole problem, learning an optimal solution much faster than a Deep Q-Learning baseline. The technique showed promise in learning the value function, suggesting that the intelligent incorporation of physics-informed invariants can help reduce the training time of machine learning tasks that seek to learn complicated nonlinear dynamics. However, these physics informed methods still want for improvement. While the Deep Q-Learning behavior may have learned far more slowly in the Cartpole example, its training behavior was more consistent.

We conclude our study by commending the designers of these physics based machine learning methods for the methods' potential, and suggest the following avenues of future research for those committed to seeing these methods receive wider adoption. Firstly, all methods, but particularly the SDE based methods, would benefit from stronger learning guarantees and convergence properties. While [16] proposed an approximation theorem (mentioned in Footnote 2 earlier), such a result is too weak to describe the rate of convergence behavior, nor does it apply for the SDE based methods. The SDE based methods in particular have higher variability, so convergence analysis would be welcome. Secondly, the implications of model misspecification for control have not been addressed adequately by research in physics informed machine learning. Finally, the success of the FBSNN suggests that intelligent decompositions of PDEs can help simplify the task of learning a complex nonlinear dynamical system, so such an avenue may yield promising alternative deep neural network architectures for similar problems.

References

[1] J. Deng, W. Dong, R. Socher, L. Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255, 2009.

[2] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25, pages 1097–1105. Curran Associates, Inc., 2012.

[3] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[4] Andrew W Senior, Richard Evans, John Jumper, James Kirkpatrick, Laurent Sifre, Tim Green, Chongli Qin, Augustin Žídek, Alexander WR Nelson, Alex Bridgland, et al. Improved protein structure prediction using potentials from deep learning. *Nature*, 577(7792):706–710, 2020.

[5] Maziar Raissi, Paris Perdikaris, and George E Karniadakis. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational Physics*, 378:686–707, 2019.

[6] Maziar Raissi, Paris Perdikaris, and George Em Karniadakis. Physics informed deep learning (part i): Data-driven solutions of nonlinear partial differential equations. *arXiv preprint arXiv:1711.10561*, 2017.

[7] Maziar Raissi, Paris Perdikaris, and George Em Karniadakis. Physics informed deep learning (part ii): Data-driven discovery of nonlinear partial differential equations. *arXiv preprint arXiv:1711.10566*, 2017.

[8] Horia Mania, Michael I. Jordan, and Benjamin Recht. Active learning for nonlinear system identification with guarantees, 2020.

[9] Sham Kakade, Akshay Krishnamurthy, Kendall Lowrey, Motoya Ohnishi, and Wen Sun. Information theoretic regret bounds for online nonlinear control, 2020.

[10] Mehryar Mohri. Reinforcement learning. Lecture Notes.

[11] Dimitri P Bertsekas. *Dynamic programming and optimal control*, volume 1.

[12] Jiongmin Yong and Xun Yu Zhou. *Stochastic controls: Hamiltonian systems and HJB equations*, volume 43. Springer Science & Business Media, 1999.

[13] E. Pardoux and S. Peng. Backward stochastic differential equations and quasilinear parabolic partial differential equations. In Boris L. Rozovskii and Richard B. Sowers, editors, *Stochastic Partial Differential Equations and Their Applications*, pages 200–217, Berlin, Heidelberg, 1992. Springer Berlin Heidelberg.

[14] Jiequn Han, Arnulf Jentzen, and Weinan E. Solving high-dimensional partial differential equations using deep learning. *Proceedings of the National Academy of Sciences*, 115(34):8505–8510, 2018.

[15] Maziar Raissi. Forward-backward stochastic neural networks: Deep learning of high-dimensional partial differential equations. *arXiv preprint arXiv:1804.07010*, 2018.

[16] Justin Sirignano and Konstantinos Spiliopoulos. Dgm: A deep learning algorithm for solving partial differential equations. *Journal of computational physics*, 375:1339–1364, 2018.

[17] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[18] Razvan V Florian. Correct equations for the dynamics of the cart-pole system. page 6.

[19] Roderic A. Grupen. Putting it all together - balancing the cart-pole. Lecture Notes.

[20] Matanya B Horowitz, Anil Damle, and Joel W Burdick. Linear hamilton jacobi bellman equations in high dimensions. In *53rd IEEE Conference on Decision and Control*, pages 5880–5887. IEEE, 2014.

A Notes on Stochastic Control

A.1 Finding the Optimal Control

As noted in the main section, the HJB PDE involves minimizing the quantity

$$\mathcal{L}(\mathbf{u}) = \left\{ q(\mathbf{x}, \mathbf{u}) + \langle \nabla_{\mathbf{x}} V, f(\mathbf{x}, \mathbf{u}) \rangle + \frac{1}{2} Tr[Hess_{\mathbf{x}} V G(t, \mathbf{x}, \mathbf{u}) G(t, \mathbf{x}, \mathbf{u})^T] \right\}$$

with respect to $\mathbf{u}$. Fortunately, since we restrict our attention to semilinear systems, this is relatively straightforward: $\nabla_{\mathbf{u}} \mathcal{L} = R\mathbf{u} + (\nabla_{\mathbf{x}} V)^T B(\mathbf{x})$. Since we want the minimum we set $\nabla_{\mathbf{u}} \mathcal{L} = \mathbf{0}$, and we find that $\mathbf{u}^* = -R^{-1} (\nabla_{\mathbf{x}} V)^T B(\mathbf{x})$. Substituting this back into the original equation yields equation (3)

A.2 Connections to Stochastic Differential Equation

In order to use the BDSE method and FBSNN methods, we need to be able to interpret equations (3) and (4) to the form specified in equation (5). This proceeds straightforwardly, and results in the following function definitions:

$$\begin{aligned}
\varphi(t, \mathbf{x}, V, \nabla_{\mathbf{x}} V) &= q(\mathbf{x}, -R^{-1} B(\mathbf{x}) \nabla_{\mathbf{x}} V) = \frac{1}{2} \left(\mathbf{x}^T Q \mathbf{x} - (\nabla_{\mathbf{x}} V)^T B(\mathbf{x}) R^{-1} B(\mathbf{x})^T (\nabla_{\mathbf{x}} V) \right) \\
\mu(t, \mathbf{x}, \nabla_{\mathbf{x}} V) &= f(\mathbf{x}, -R^{-1} B(\mathbf{x}) \nabla_{\mathbf{x}} V) = a(\mathbf{x}) - B(\mathbf{x})(-R^{-1} B(\mathbf{x})^T \nabla_{\mathbf{x}} V) \\
G(t, \mathbf{x}, \mathbf{u}) &= \sigma I
\end{aligned}$$

A.3 Architecture of the Deep Galerkin Method

Sirignano et. al. propose a rather complicated deep network architecture, composed of LSTM like layers [16]. We describe them fully below, although we direct the reader to [16] for a more detailed explanation.

$$\begin{aligned}
S^1 &= \sigma(W^1 \vec{x} + b^1), \\
Z^l &= \sigma(U^{z,l} \vec{x} + W^{z,l} S^l + b^{z,l}), \qquad l = 1, ..., L, \\
G^l &= \sigma(U^{g,l} \vec{x} + W^{g,l} S^1 + b^{g,l}), \qquad l = 1, ..., L, \\
R^l &= \sigma(U^{r,l} \vec{x} + W^{r,l} S^l + b^{r,l}), \qquad l = 1, ..., L, \\
H^l &= \sigma(U^{h,l} \vec{x} + W^{h,l} (S^l \odot R^l) + b^{h,l}), \qquad l = 1, ..., L, \\
S^{l+1} &= (1 - G^l) \odot H^l + Z^l \odot S^l, \qquad l = 1, ..., L, \\
f(t, x; \theta) &= W S^{L+1} + b,
\end{aligned}$$

where $\vec{x} = (t, x)$, the number of hidden layers is $L + 1$, and $\odot$ denotes element-wise multiplication (i.e., $z \odot v = (z_0 v_0, ..., z_N v_N)$). The parameters are

$$\theta = \{W^1, b^1, (U^{z,l}, W^{z,l}, b^{z,l})_{l=1}^L, (U^{g,l}, W^{g,l}, b^{g,l})_{l=1}^L, (U^{r,l}, W^{r,l}, b^{r,l})_{l=1}^L, (U^{h,l}, W^{h,l}, b^{h,l})_{l=1}^L, W, b\}.$$

The number of units in each layer is M and $\sigma : \mathbb{R}^M \to \mathbb{R}^M$ is an element-wise nonlinearity:

$$\sigma(z) = (\phi(z_1), \phi(z_2), ..., \phi(z_M)),$$

where $\phi : \mathbb{R} \to \mathbb{R}$ is a nonlinear activation function such as the tanh function, sigmoidal function or ReLU. The parameters in θ have dimensions $W^1 \in \mathbb{R}^{M \times (d+1)}$, $b^1, b^{z,l}, b^{g,l}, b^{r,l}, b^{h,l} \in \mathbb{R}^M$, $U^{z,l}, U^{g,l}, U^{z,l}, U^{h,l} \in \mathbb{R}^{M \times (d+1)}$, $W^{z,l}, W^{g,l}, W^{r,l}, W^{h,l} \in \mathbb{R}^{M \times M}$, $W \in \mathbb{R}^{1 \times M}$, and $b \in \mathbb{R}$

B Experiments

B.1 Code and Environments

All *LQG* examples were trained on an Nvidia GTX 1070 for equivalent comparison. Code was adapted and added from initial papers, and is available at the following repositories. The cartpole specific dynamics are included in the FBSNN repo.

- Deep Galerkin Method (https://github.com/Leo-Enrique-Wu/DGM)
- Deep BDSE Method (https://github.com/frankhan91/DeepBSDE)
- FBSNN (also includes Cartpole Simulation) (https://github.com/nitinshyamk/FBSNNs)

B.2 Semi analytical solution for the Linear Quadratic Gaussian problem

The semi analytical solution to the proposed Linear Quadratic Gaussian problem from Section 5.1 is discussed briefly here. For evaluation to compare the accuracy, we compute $u(t, \mathbf{x})$ using Monte Carlos sampling 10k samples for each point. This is tractable with tensorized operations. We also used an upper bound to sanity check our results.

$$
\begin{aligned}
u(t,x) &= \frac{-1}{\lambda} \ln \mathbb{E}\left[e^{-\lambda g(x+\sqrt{2}W_{T-t})}\right] \\
&= \frac{-1}{\lambda} \ln \mathbb{E}\left[\left(\frac{1+||x+\sqrt{2}W_{T-t}||^2}{2}\right)^{-\lambda}\right] \\
&= \frac{-1}{\lambda} \ln \mathbb{E}\left[\left(\frac{1+\sum_{i=1}^n Y_i}{2}\right)^{-\lambda}\right]
\end{aligned}
$$

Here, Y_i is the noncentral $\chi_1^2\left(\frac{x_i^2}{2(T-t)^2}\right)$ distribution, which has has mean $2(T-t)^2 + x_i^2$. Apply Jensen's inequality since $f(x) = x^{-\lambda}$ is convex for $x > 0$ and positive λ.

$$
\begin{aligned}
\mathbb{E}\left[\left(\frac{1+\sum_{i=1}^n Y_i}{2}\right)^{-\lambda}\right] &\geq \mathbb{E}\left[\frac{1+\sum_{i=1}^n Y_i}{2}\right]^{-\lambda} \\
&= \left(\frac{1}{2} + \frac{1}{2}||\mathbf{x}||^2 + n(T-t)^2\right)^{-\lambda}
\end{aligned}
$$

This upper bound is substituted back in, and since ln is an increasing function and we have a negative factor, the lower bound on the expectation translates to an upper bound on $u(t, x)$:

$$
\begin{aligned}
u(t,x) &= \frac{-1}{\lambda} \ln \mathbb{E}\left[\left(\frac{1+\sum_{i=1}^n Y_i}{2}\right)^{-\lambda}\right] \\
&\leq \ln\left(\frac{1}{2} + \frac{1}{2}||\mathbf{x}||^2 + n(T-t)^2\right)
\end{aligned}
$$

B.3 Hyperparameter Optimization for Cartpole

For all hyperparameters, we set $T = 1.0, N = 50$ and $Q = Qt = I$ and $R = 1$. We sought to optimize the network architecture over a few different ways.

For the FBSNN method, we initially used a batch size of $K = 5000$ and sought to optimize the layer size.

Layer Size	8	16	32	64	128	256
Loss (averaged per sample)	79.68	2.89	0.124	0.772	62.225	66.14

For using the DGM method to solve the Cart-Pole problem, we searched for several different hyperparameter settings for the neural network, but the losses decrease slowly, and even in the best cases, retained an absolute value far too high for effective control.

# of layers	nodes per layer	learning rate	the loss after 1000 iter	initial loss	decreasing factor
3	50	0.0001	$9.68 \cdot 10^7$	$1.17 \cdot 10^8$	1.2
3	50	0.001	$4.62 \cdot 10^7$	$1.00 \cdot 10^8$	2.2
3	50	0.01	$4.59 \cdot 10^7$	$1.08 \cdot 10^8$	2.4
3	50	0.1	$6.06 \cdot 10^7$	$7.74 \cdot 10^7$	1.3
5	50	0.01	$5.15 \cdot 10^7$	$1.33 \cdot 10^8$	2.6
5	50	0.01	$2.57 \cdot 10^7$	$1.09 \cdot 10^8$	4.3

* Note:

The decreasing factor $= \frac{loss_0}{loss_{1000}}$