Euro. Jnl of Applied Mathematics (2021), *vol.* 32, *pp.* 515–539 © The Author(s), 2020. Published by Cambridge University Press.
doi:10.1017/S0956792520000327

Deep learning models for global coordinate transformations that linearise PDEs

CRAIG GIN[1], BETHANY LUSCH[2], STEVEN L. BRUNTON[1,3] and J. NATHAN KUTZ[1]

[1] *Department of Applied Mathematics, University of Washington, Seattle, WA 98195, USA*
emails: crgin@ncsu.edu; kutz@uw.edu
[2] *Argonne Leadership Computing Facility, Argonne National Laboratory, Lemont, IL 60439, USA*
email: blusch@anl.gov
[3] *Department of Mechanical Engineering, University of Washington, Seattle, WA 98195, USA*
email: sbrunton@uw.edu

(*Received 6 November 2019*; *revised 3 July 2020*; *accepted 18 August 2020*; *first published online 24 September 2020*)

We develop a deep autoencoder architecture that can be used to find a coordinate transformation which turns a non-linear partial differential equation (PDE) into a linear PDE. Our architecture is motivated by the linearising transformations provided by the Cole–Hopf transform for Burgers' equation and the inverse scattering transform for completely integrable PDEs. By leveraging a residual network architecture, a near-identity transformation can be exploited to encode intrinsic coordinates in which the dynamics are linear. The resulting dynamics are given by a Koopman operator matrix **K**. The decoder allows us to transform back to the original coordinates as well. Multiple time step prediction can be performed by repeated multiplication by the matrix **K** in the intrinsic coordinates. We demonstrate our method on a number of examples, including the heat equation and Burgers' equation, as well as the substantially more challenging Kuramoto–Sivashinsky equation, showing that our method provides a robust architecture for discovering linearising transforms for non-linear PDEs.

Key words: Koopman theory, deep neural networks, residual networks, linearising transforms, Cole–Hopf transform

2020 Mathematics Subject Classification: Primary: 35A22. Secondary: 35A35; 37M99; 65P99; 68T99.

1 Introduction

Partial differential equations (PDEs) provide a theoretical framework for modelling spatio-temporal systems across the biological, physical and engineering sciences. Analytic solution techniques are readily available for PDEs that are linear and have constant coefficients [14]. These PDEs include canonical models, such as the heat equation, wave equation and Laplace's equation which are amenable to standard separation of variable techniques and linear superposition. In contrast, there is no general mathematical architecture for solving non-linear PDEs as methods like separation of variables fail to hold, thus recourse to computational solutions is necessary. There are a few, but notable, exceptions: (i) the Cole–Hopf transformation [8, 16] for solving diffusively regularised Burgers' equation and (ii) the *inverse scattering transform* (IST) [1] for solving a class of completely integrable PDEs, such as Korteweg–de Vries, nonlinear Schrödinger, and Klein–Gordon. The success of Cole–Hopf and IST is achieved by providing a *linearising* transformation of the governing non-linear equations. Thus, these methods provide a transformation to a new coordinate system where the dynamics are characterised by a linear PDE and for which recourse can be made to the well-established linear methods for estimation, control, and uncertainty quantification. Recent efforts have shown that it is possible to use neural networks to discover advantageous coordinate transformations for dynamical systems [7, 10, 24, 27, 29, 36, 42, 43, 47]. Indeed, such architectures provide a strategy to discover linearising transformations, such as the Cole–Hopf and IST. In this manuscript, we develop a principled method for using neural networks for discovering coordinate transformations for linearising non-linear PDEs, demonstrating that transformations such as Cole–Hopf can be discovered from data alone with proper architecting of the network.

Linearising transforms fall broadly under the aegis of Koopman operator theory [20] which has a modern interpretation in terms of dynamical systems theory [6,30–32]. Koopman operators can only be constructed explicitly in limited cases [4]; however, *dynamic mode decomposition* (DMD) [41] provides a numerical algorithm to provide an approximation to the Koopman operator [40], with many recent extensions improving on the DMD approximation [22]. Importantly, many of the advantageous transformations highlighted above attempt to construct Koopman embeddings for the dynamics using neural networks [24, 27, 29, 36, 42, 43, 47]. This is in addition to enriching the observables of DMD [19, 23, 34, 35, 37, 45, 46]. Thus, neural networks have emerged as a highly effective mathematical architecture for approximating complex data [2, 13]. Their universal approximation properties [9, 17] are ideal for learning the coordinate transformations required for linearising non-linear PDEs. Neural networks have also been used in this context to discover time-stepping algorithms for complex systems [12, 39]. This is a slightly different but related task to learning linearising transforms, which is what is advocated here. The advantage of learning a linearising coordinate transformation is that the vast arsenal of tools that have been developed for linear systems can be used for non-linear dynamical systems. For example, in many applications, we want to not just predict the future of a system but also control its behaviour. The theory of controls for linear systems is much more well developed than that for non-linear systems. In a similar vein, we may have incomplete information about the state of a high-dimensional system and need to estimate the state from limited sensor measurements. This is more tractable for linear systems. Finally, a transformation to a linear system provides a simplified framework for uncertainty quantification for highly non-linear and chaotic systems for which modelling uncertainty is challenging.

Critical to the success of a neural network is imposing proper constraints, or regularisers, for the desired task. For spatio-temporal systems modelled by PDEs, there are a number of constraints that are required for success. These constraints are largely motivated by domain knowledge, thus producing a physics-informed machine learning architecture for PDEs. Specifically, we identify four critical components for successfully training a neural network for non-linear PDEs: (i) the appropriate neural network architecture for the desired task, (ii) a method for handling the identity transformation as ideas of near-identity transformation are critical, (iii) domain knowledge constraints for the PDEs and (iv) judiciously chosen spatio-temporal data for training the network.

Imposing structure on the neural network is the first critical design task. As shown in recent works, autoenconders (AEs) are a physics-informed choice as they have been shown to be able to take input data from the original high-dimensional input space to the new coordinates which are at the intrinsic rank of the underlying dynamics [7, 27, 38]. AEs allow for the computation of a non-linear dimensionality reduction in constructing a linear reduced-order model. The AE

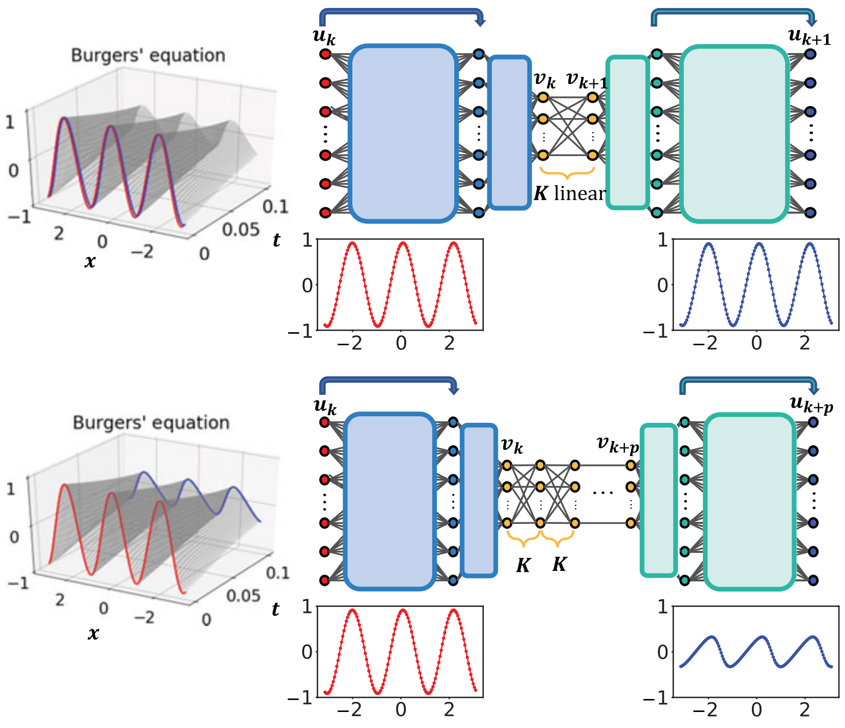

FIGURE 1. A deep autoencoder is used to find coordinate transformations to linearise PDEs. The encoder finds a set of intrinsic coordinates for which the dynamics are linear. Then the dynamics are given by a matrix **K**. The decoder transforms back to the original coordinates. Multiple time step prediction can be performed by repeated multiplication by the matrix **K** in the intrinsic coordinates.

mapping is significantly improved if the identity map can be easily implemented. The success of near-identity transformations for transforming dynamical systems motivates the leading order identity mapping [33, 44]. As will be shown, non-linear activation functions such as the rectified linear unit (ReLU) make it difficult to produce the identity operator. The success of *deep residual networks* (DRN) [15] motivate our approach to handling the identity map. We have found that the AE architecture which leverages the DRN structure provides a physics-informed architecture that leverages the concepts of near-identity transforms along with the low-dimensional intrinsic rank of the physics itself.

Finally, the data must be judiciously chosen. As stated by S. Mallat: *Supervised learning is a high-dimensional interpolation problem* [28]. Thus, a wide range of initial conditions and trajectories must be sampled in order to construct a robust and accurate linearising transform. The success of training the AE, and indeed of any neural network architecture, hinges on training with a broad enough class of data so that new spatio-temporal trajectories can leverage the interpolation powers of the AE infrastructure. We emphasise again that neural networks are interpolation engines and fail in extrapolatory scenarios.

We demonstrate our method on a set of prototype PDE models: the heat equation, Burgers' equation and the Kuramoto–Shivasinsky (KS) equation. Figure 1 illustrates the overall approach. For the first two equations, we have ground truth analytic results to validate the methodology. For the KS equation, we provide a first mapping of the PDE to a linearised system which can be completely inverted, although a number of Galerkin schemes have been advocated to characterise the KS dynamics [11]. These examples demonstrate the power of the methodology and the required regularisations necessary for training neural networks for learning coordinate embeddings.

2 Numerical time-stepping and dynamics

Given the spatio-temporal nature of data generated by PDEs, we develop neural network architectures that exploit various aspect of numerical time-stepping schemes. This begins with understanding the near-identity transformation generated from numerical discretisation schemes for solving PDEs.

2.1 Identity functions and neural networks

A feed-forward neural network $f(x)$ is a composition of functions of the form:

$$f(x) = f_n(\cdot; W_n, b_n) \circ \cdots \circ f_2(\cdot; W_2, b_2) \circ f_1(x; W_1, b_1), \tag{2.1}$$

where each function f_j is an affine transformation, parameterised by the weights W_j and the biases b_j, followed by a non-linear activation function. Although training a neural network to represent the identity function $g(x) = x$ may seem simple, we show that non-linear activation functions such as ReLU make this non-trivial. We consider a one-dimensional data set x composed of random real numbers from $[-1, 1]$. We assign each point a label $y = x$. Then, we create a small seven-parameter neural network with a one-node input layer, two-node hidden layer with a ReLU activation and a one-node linear output layer. Each node in the hidden layer and output layer have a bias term. This network is depicted in Figure 2 and is given by the formula:

$$f(x) = W_2 \sigma(W_1 x + b_1) + b_2, \tag{2.2}$$

where $x \in \mathbb{R}$, $W_1 \in \mathbb{R}^{2\times 1}$, $b_1 \in \mathbb{R}^2$, $W_2 \in \mathbb{R}^{1\times 2}$, $b_2 \in \mathbb{R}$ and $\sigma(t) = \max\{0, t\}$ is the ReLU activation function. The red network in Figure 2 is labelled with parameters that would map input x to output $y = x$ perfectly for any $x \in \mathbb{R}$. However, when we train this network on our training data from $[-1, 1]$, different parameters are chosen, such as those shown on the blue network in Figure 2. This trained network is accurate in the training domain of $[-1, 1]$ but diverges from $g(x) = x$ outside of that domain. Further, as demonstrated in Figure 3, if we train the network repeatedly with different random initialisations of the parameters, we obtain different results. In some cases (bottom-left and bottom-right of Figure 3), the training loss plateaus and the network fails to accurately fit the training data at all. This is due to the tendency of deep and narrow ReLU networks to collapse to the mean value of the function [25, 26]. We have two observations from this experiment:

(1) We are reminded that neural networks cannot be relied upon to extrapolate outside of the training domain.
(2) Even though it is possible to represent the identity function with this non-linear neural network, it is non-trivial for the training algorithm to fit the data.

This latter observation contributes to our motivation to use a residual neural network structure. Although in theory a near-identity function $g(x)$ may be representable by a neural network, the optimisation may be more successful at learning the residual $h(x)$ where $g(x) = x + h(x)$.

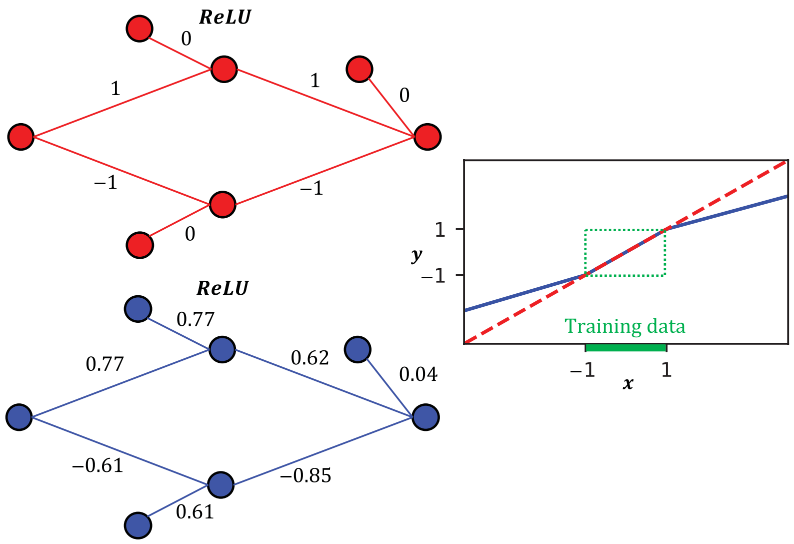

FIGURE 2. We train a seven-parameter non-linear neural network to fit $g(x) = x$ with data from $[-1, 1]$. Although there is an optimal solution (see the red network) that is valid for all $x \in \mathbb{R}$, the training algorithm chooses parameters such as those labelled on the blue network. On the right, the function learned by the blue network is plotted in blue. This is compared to the identity function, plotted in red. The training domain is annotated in green. We note that the trained network should not be used for extrapolation.

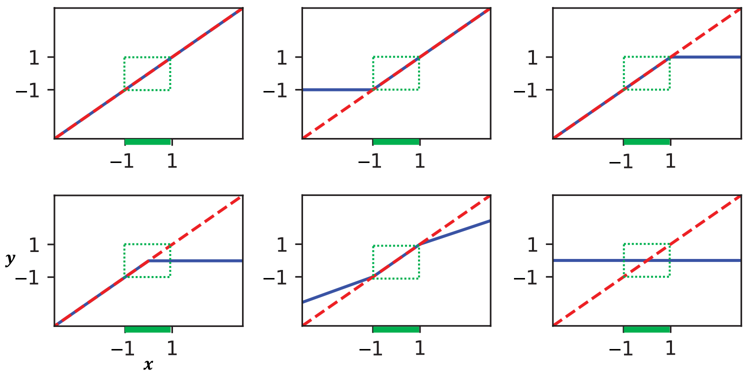

FIGURE 3. We train a small neural network (shown in Figure 2) to fit $g(x) = x$ with data from $[-1, 1]$. We show results from six training trials with different random initialisations. In each of the six plots, the function learned by the network is plotted in blue. This is compared to the identity function, plotted in red. The training domain is annotated in green. We note that the trained network should not be used for extrapolation, and that sometimes the trained network is not accurate even on the training data.

2.2 Building networks for time-stepping dynamics

In what follows, we consider several PDEs, typically non-linear, of the form:

$$u_t = F(u, u_x, u_{xx}, \ldots, x, t), \tag{2.3}$$

where $F(\cdot)$ characterises the governing equations. When discretised in space and time, the PDE becomes a finite-dimensional, discrete-time dynamical system of the form [21]:

$$\mathbf{u}_{k+1} = \mathbf{F}(\mathbf{u}_k), \tag{2.4}$$

where $\mathbf{u}_k$ is a spatial discretisation of the function $u(x, t)$ at time t_k where $t_k = k\Delta t$ for some time step Δt. In the case that the function $\mathbf{F}$ is linear, the future values of the state $\mathbf{u}$ can be solved for exactly using a spectral expansion. However, $\mathbf{F}$ is typically non-linear, and there is no general framework for solving non-linear systems.

Koopman operator theory provides a means to linearise a non-linear dynamical system. Let $X \subseteq \mathbb{R}^n$ denote the state space of the dynamical system (2.4). The Koopman operator, $\mathcal{K}$, is a linear operator that acts on the space of observables $g : X \to \mathbb{R}$. There are different choices for the space of observables, but a typical choice is the Hilbert space of Lebesgue square-integrable functions. Most importantly, it is an infinite-dimensional space. The Koopman operator, or composition operator, is defined by

$$\mathcal{K}g = g \circ \mathbf{F}. \tag{2.5}$$

On a trajectory of the dynamical system,

$$\mathcal{K}g(\mathbf{u}_k) = g \circ \mathbf{F}(\mathbf{u}_k) = g(\mathbf{u}_{k+1}). \tag{2.6}$$

Therefore, the Koopman operator advances the observables in time. Because of its linearity, the behaviour of the Koopman operator is completely determined by its eigenvalues and eigenfunctions. A Koopman eigenfunction φ corresponding to eigenvalue λ satisfies

$$\mathcal{K}\varphi = \lambda\varphi. \tag{2.7}$$

An arbitrary observable can be expanded in terms of the Koopman eigenfunctions assuming the eigenfunctions form a basis of the space of observables:

$$g = \sum_{j=1}^{\infty} a_j \varphi_j. \tag{2.8}$$

Applying the Koopman operator to this expansion gives an easy way to represent the dynamics of the observable g on a trajectory of the dynamical system:

$$g(\mathbf{u}_{k+1}) = \mathcal{K}g(\mathbf{u}_k) = \sum_{j=1}^{\infty} \lambda_j a_j \varphi_j(\mathbf{u}_k). \tag{2.9}$$

Such spectral decompositions are standard for linear differential equations [3].

Although the Koopman operator acts on an infinite-dimensional space, we can obtain a finite-dimensional approximation by considering the space spanned by finitely many Koopman eigenfunctions. Acting on this space, the Koopman operator is just a matrix. Therefore, Koopman operator theory provides an approach to find an intrinsic coordinate system in which the dynamical system has linear dynamics. However, finding a useful set of approximate Koopman eigenfunctions may be arbitrarily challenging. In fact, they may be irrepresentable [5]. Additionally, using a finite-dimensional representation may result in closure issues [4]. In order to overcome these challenges, we can use the ability of neural networks to accurately approximate arbitrary functions [17].

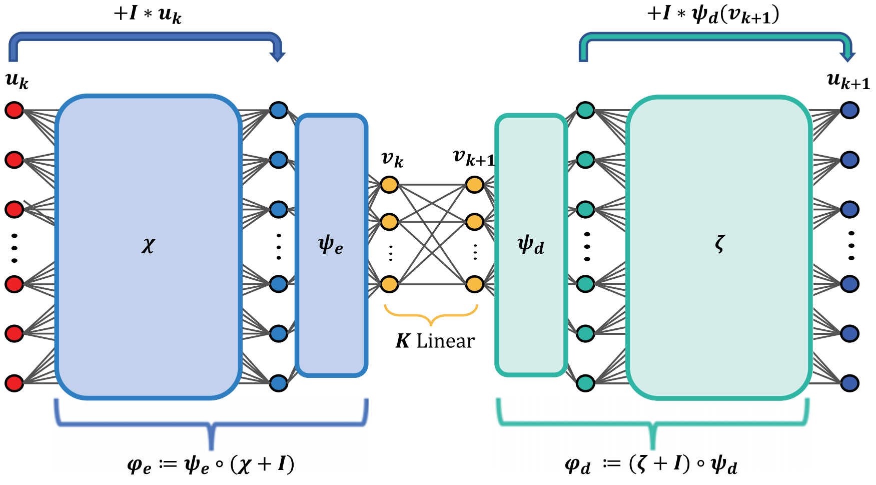

FIGURE 4. The network architecture used to find the Koopman eigenfunctions. The network consists of an outer encoder $\chi + \mathbf{I}$, inner encoder ψ_e, dynamics matrix $\mathbf{K}$, inner decoder ψ_d and outer decoder $\zeta + \mathbf{I}$. The outer encoder and decoder use a residual neural network architecture. The encoder φ_e and decoder φ_d are constrained by the loss function to be inverses. The same is true of the outer encoder and decoder and inner encoder and decoder.

In this work, we use the universal approximation properties of neural networks to find such linearising coordinate transformations. The neural network $f(\mathbf{u})$ advances the state variable forward in time:

$$\mathbf{u}_{k+1} = f(\mathbf{u}_k) \tag{2.10}$$

and can be expressed by the formula:

$$f(\mathbf{u}) = \varphi_d(\mathbf{K}(\varphi_e(\mathbf{u}))). \tag{2.11}$$

The network architecture is shown in Figure 4. The input of the the network $\mathbf{u}_k$ is the state vector at time t_k and the output is the state vector at time t_{k+1}. The network consists of three parts: (i) the encoder φ_e, (ii) the linear dynamics $\mathbf{K}$ and (iii) the decoder φ_d. Both the encoder and decoder are split into two parts. The encoder consists of the outer encoder $\chi + \mathbf{I}$ and the inner encoder ψ_e:

$$\varphi_e(\mathbf{u}) = \psi_e((\chi + \mathbf{I})(\mathbf{u})). \tag{2.12}$$

The outer encoder performs a coordinate transformation into a space in which the dynamics are linear. The inner encoder either (i) diagonalises the system, (ii) reduces the dimensionality or (iii) both. The decoder consists of the inner decoder ψ_d and the outer decoder $\zeta + \mathbf{I}$:

$$\varphi_d(\mathbf{u}) = (\zeta + \mathbf{I})(\psi_d(\mathbf{u})). \tag{2.13}$$

The inner and outer decoder are approximate inverses of the inner and outer encoder, respectively.

For a sufficiently small time step, the function $\mathbf{F}(\mathbf{u})$ in equation (2.4) is a near-identity function, which is a simple representation of an integration scheme [12, 21, 38, 39]. As demonstrated in the previous section, the identity transformation can be difficult to learn using non-linear neural networks. Therefore, special care must be given to account for the fact that the network is

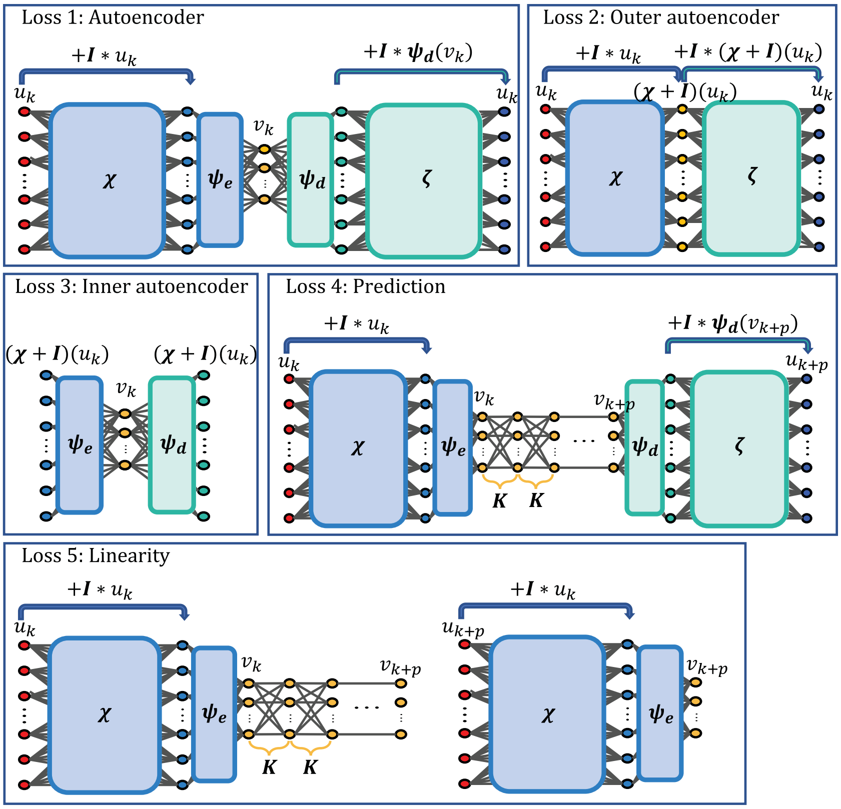

FIGURE 5. A depiction of the loss functions used for training the neural network.

approximating a near-identity function [33, 44]. This is handled in two ways. First, a residual neural network architecture is used for both the outer encoder and outer decoder [15]. The coordinate transformation given by the outer encoder is represented as the identity transformation plus a residual which is given by the neural network χ. Therefore, the outer encoder is $\chi + \mathbf{I}$. Similarly, the outer decoder is $\zeta + \mathbf{I}$ where ζ is a neural network. Second, the interior layers of the network, ψ_e, $\mathbf{K}$, and ψ_d, are all initialised as identity or identity-like matrices (see Appendix A).

The loss function used to train the network is the sum of five different losses and a regularisation term:

$$L = L_1 + L_2 + L_3 + L_4 + L_5 + R. \tag{2.14}$$

In the examples that follow, the data consist of discretised solutions to PDEs with N different initial conditions and solved M steps forward in time. Let $\mathbf{u}_k^j$ denote the solution with initial condition indexed by j and time step indexed by k where $k = 0$ denotes an initial condition. The five loss functions are depicted in Figure 5 and given by formulas below. Each loss enforces a desired condition:

(1) **Loss 1: Autoencoder loss**. We want an invertible transformation between the state space and intrinsic coordinates for which the dynamics are linear. The transformation into the intrinsic coordinates is given by the encoder φ_e and the transformation back into the state space is

given by the decoder φ_d. Therefore, we wish for the autoencoder $\varphi_d \circ \varphi_e$ to reconstruct the inputs of the network as closely as possible. This autoencoder loss is given by a relative mean squared error where the mean is taken over all initial conditions and all time steps:

$$L_1 = \frac{1}{N}\sum_{j=1}^{N}\frac{1}{M+1}\sum_{k=0}^{M}\frac{\left\|\mathbf{u}_k^j - \varphi_d(\varphi_e(\mathbf{u}_k^j))\right\|_2^2}{\left\|\mathbf{u}_k^j\right\|_2^2}, \tag{2.15}$$

where $\|\cdot\|_2$ denotes the 2-norm.

(2) **Loss 2: Outer autoencoder loss**. We wish for the outer encoder and the outer decoder to form an autoencoder. This separates the transformation into the intrinsic coordinates from dimensionality reduction and/or diagonalisation which are performed by the inner encoder/decoder. Disentangling these two processes allows for better interpretability in the case of Burgers' equation in Section 4. The outer autoencoder loss is given by

$$L_2 = \frac{1}{N}\sum_{j=1}^{N}\frac{1}{M+1}\sum_{k=0}^{M}\frac{\left\|\mathbf{u}_k^j - (\zeta + \mathbf{I})((\chi + \mathbf{I})(\mathbf{u}_k^j))\right\|_2^2}{\left\|\mathbf{u}_k^j\right\|_2^2}. \tag{2.16}$$

(3) **Loss 3: Inner autoencoder loss**. The inner autoencoder loss is given by

$$L_3 = \frac{1}{N}\sum_{j=1}^{N}\frac{1}{M+1}\sum_{k=0}^{M}\frac{\left\|(\chi + \mathbf{I})\mathbf{u}_k^j - \psi_d(\psi_e((\chi + \mathbf{I})(\mathbf{u}_k^j)))\right\|_2^2}{\left\|(\chi + \mathbf{I})\mathbf{u}_k^j\right\|_2^2}. \tag{2.17}$$

(4) **Loss 4: Prediction loss**. The output of the network should accurately predict the state $\mathbf{u}_{k+1}$ when given the state at the previous time $\mathbf{u}_k$. For a given example, the prediction loss is given by $\|\mathbf{u}_{k+1} - \varphi_d(\mathbf{K}\varphi_e(\mathbf{u}_k))\|_2^2$. Furthermore, we would like to be able to predict multiple time steps into the future by iteratively multiplying by the matrix $\mathbf{K}$. Therefore, in general, we have $\left\|\mathbf{u}_{k+p} - \varphi_d(\mathbf{K}^p\varphi_e(\mathbf{u}_k))\right\|_2^2$. Note that multi-step prediction is done by evolving multiple steps in the intrinsic coordinates and not by passing in and out of the intrinsic coordinates at each time step. The prediction loss is obtained using the initial condition to predict all future time steps:

$$L_4 = \frac{1}{N}\sum_{j=1}^{N}\frac{1}{M}\sum_{p=1}^{M}\frac{\left\|\mathbf{u}_p^j - \varphi_d(\mathbf{K}^p\varphi_e(\mathbf{u}_0^j)\right\|_2^2}{\left\|\mathbf{u}_p^j\right\|_2^2}. \tag{2.18}$$

(5) **Loss 5: Linearity loss**. The dynamics in the intrinsic coordinates should be linear. Therefore, we enforce a prediction loss within these coordinates:

$$L_5 = \frac{1}{N}\sum_{j=1}^{N}\frac{1}{M}\sum_{p=1}^{M}\frac{\left\|\varphi_e(\mathbf{u}_p^j) - \mathbf{K}^p\varphi_e(\mathbf{u}_0^j)\right\|_2^2}{\left\|\varphi_e(\mathbf{u}_p^j)\right\|_2^2}. \tag{2.19}$$

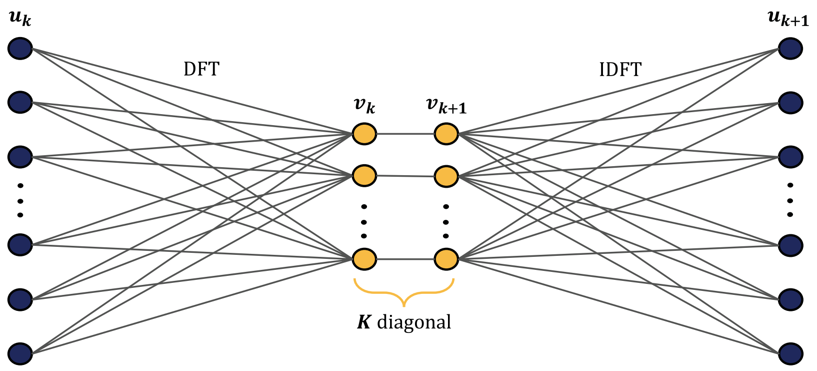

FIGURE 6. The network architecture for the heat equation. The input and output layers have $n = 128$ neurons and the two hidden layers have 21 neurons. The network has no activation functions.

(6) **Regularisation**. All weight matrices except for **K** are regularised with ℓ^2 regularisation:

$$R = \lambda \sum_i \|\mathbf{W}_i\|_F , \tag{2.20}$$

where λ is a regularisation factor, the $\mathbf{W}_i$ are the weight matrices, the sum is over all weight matrices in the neural network and $\|\cdot\|_F$ is the Frobenius norm.

The data for training the neural networks are created by performing numerical simulations of the given PDE. The initial conditions used and discretisation details are described for each example below. In each case, we choose the data to be sufficiently diverse as to learn a global coordinate transformation that holds for a wide variety of initial conditions. Note that our approach is completely data-driven – no knowledge of the underlying equations is needed. Therefore, it can be used for experimental data for which the governing equations are unknown.

3 Heat equation

The first PDE that we consider is the one-dimensional heat equation:

$$u_t = u_{xx}, \qquad x \in (-\pi, \pi), \tag{3.1}$$

with periodic boundary conditions. We discretise the spatial domain using $n = 128$ equally spaced points.

Because the heat equation is linear, finding the Koopman eigenfunctions amounts to diagonalising the heat equation. The network architecture that we use for the heat equation is shown in Figure 6. There is no outer encoder or decoder because there is no need for a linearising transformation. Therefore, the entire network is linear and consists of two hidden layers. The widths of the input and output layers match the spatial discretisation: $n = 128$ neurons in each. The width of the hidden layers is a parameter r that can be adjusted in order to obtain a reduced-order model of rank r for the heat equation. We chose $r = 21$. The heat equation is linearised by the Fourier transform so the encoder should mimic a discrete Fourier transform (DFT) that is truncated to include the r most dominant modes. The decoder plays the role of the inverse DFT. The matrix **K** that represents the dynamics is forced to be diagonal.

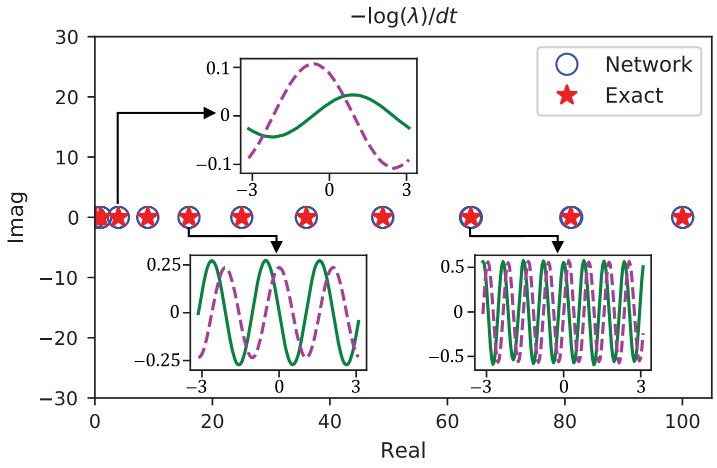

FIGURE 7. The eigenvalues of the matrix **K** from the neural network are plotted along with the exact, discrete-time eigenvalues of the heat equation. The eigenfunctions corresponding to several eigenvalues are shown.

The training data consist of 8000 trajectories from the heat equation. The initial condition in all cases is randomly created white noise and therefore has no particular structure. The trajectories consist of 50 equally spaced time steps with $\Delta t = 0.0025$. The validation data have the same structure but contain 2000 trajectories. The heat equation was solved using a spectral method.

For the heat equation, the discrete-time eigenvalues are

$$\lambda = e^{-\omega^2 \Delta t}, \qquad \omega = 0, \pm 1, \pm 2, \ldots \tag{3.2}$$

Because the spatial discretisation is $n = 128$ points, ω ranges from -64 to 63. The eigenfunctions are $\sin(\omega x)$ and $\cos(\omega x)$ for each positive value of ω. Because the high-frequency waves decay faster than the low-frequency waves, we expect the 21×21 matrix K to have the eigenvalues:

$$\lambda = e^{-\omega^2 \Delta t}, \qquad \omega = 0, \pm 1, \pm 2, \ldots, \pm 10, \tag{3.3}$$

and therefore the eigenvalues satisfy

$$-\log(\lambda)/\Delta t = \omega^2, \qquad \omega = 0, \pm 1, \pm 2, \ldots, \pm 10. \tag{3.4}$$

The eigenvalues of the network are shown in Figure 7 along with the exact analytical values given by equation (3.4). Note that for $\omega \neq 0$, the eigenvalues that are plotted occur in pairs. There is very good agreement between the eigenvalues from the neural network and the exact values. Figure 7 also shows the eigenfunctions corresponding to $\omega = \pm 1, \pm 3, \pm 8$. These were plotted by taking the eigenvectors of **K** and feeding them through the decoder. As expected, the eigenfunctions are oscillatory and have a frequency that corresponds to the eigenvalues they are associated with. However, note that the eigenfunctions do not all have the same amplitude. This is because the DFT is not a unique transformation to diagonalise the heat equation. In addition to performing a DFT, the encoder of the network can also scale each of the Fourier modes by any constant. The decoder must then invert the scalings performed by the encoder. The fact that the encoder differs from the DFT is shown in Figure 8 which compares the result of feeding a particular function through the encoder versus the DFT of that function.

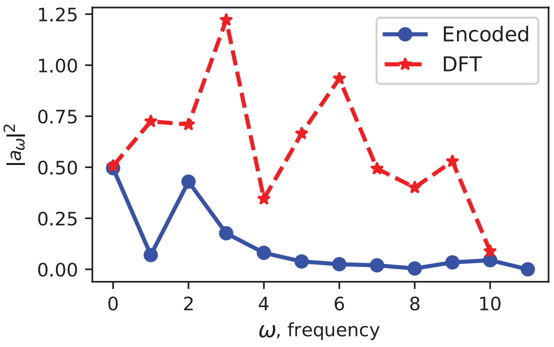

FIGURE 8. For a particular function, the DFT and the encoded function are shown. The encoder is an alternate transformation to diagonalise the heat equation.

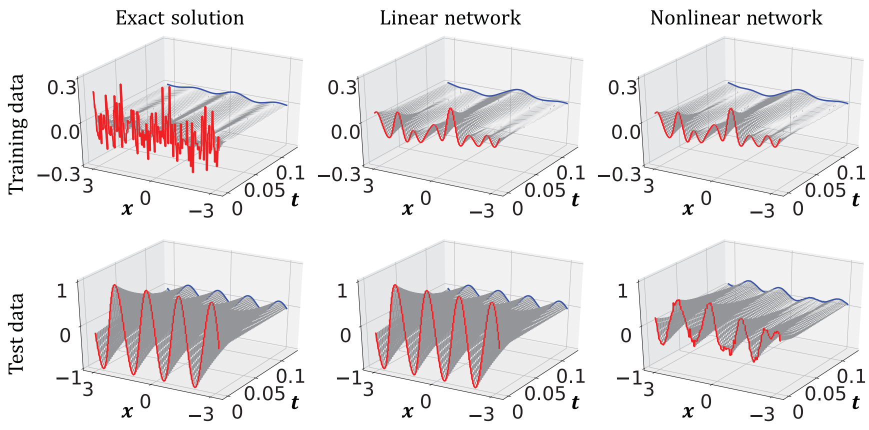

FIGURE 9. Plots of (a) an exact solution to the heat equation, (b) the prediction given by the linear neural network and (c) the prediction given by a non-linear neural network. The top row shows an example from the training data for which the initial condition is white noise. The bottom row shows an example of test data for which the initial condition is $u_0(x) = \sin(4x)$. For the prediction at the initial time $t = 0$, the initial condition is encoded and then decoded so the red curve gives an indication of the performance of the autoencoder.

Because the entire network is linear, having the correct eigenvalues and eigenvectors is enough to have a global representation of the heat equation. Therefore, it can be used for prediction for initial conditions that are not represented in the training data. As an example, the network was used to do prediction for the initial condition $u_0(x) = \sin(4x)$. The exact solution and the output of the network are shown in Figure 9 as the two leftmost plots on the bottom row. The two are indistinguishable. Note that for all network predictions that follow, only the initial condition is needed. To get a prediction at time $t_p = p\Delta t$, the initial condition is encoded, multiplied by the matrix $\mathbf{K}^p$, and then decoded. For the prediction at the initial time $t = 0$, the initial condition is encoded and then decoded so the red curve gives an indication of the performance of the autoencoder.

It is very important that we leveraged the fact that the heat equation is linear and used a linear neural network. This allowed for the ability to generalise outside of the training data. For contrast, we also trained a non-linear network by adding two fully connected layers, the first of which has a ReLU activation function, to both the encoder and decoder. The total validation loss for the non-linear network is 0.0449 which is less than the 0.0711 validation loss for the linear network. The top row of Figure 9 shows the predictions given by both networks for one trajectory in the training data. The linear and non-linear networks have similar accuracy. However, prediction using the non-linear network for data with a different structure than the training data is very poor, as evidenced by the rightmost plot of the bottom row in Figure 9.

Although the heat equation is already linear, it serves as a useful first example because the network was able to find the eigenvalues and eigenfunctions of the heat equation just from the data. In addition, it was able to identify the optimal reduced-order model of rank $r = 21$.

4 Burgers' equation

For our second example, we consider the non-linear PDE known as the diffusively regularised Burgers' equation which can be written as:

$$u_t + \epsilon u u_x = \mu u_{xx}, \qquad x \in (-\pi, \pi), \tag{4.1}$$

where the parameter ϵ is the strength of advection and the parameter μ is the strength of diffusion. For our results, we use $\epsilon = 10$ and $\mu = 1$. We again use periodic boundary conditions and discretise the spatial domain with $n = 128$ equally spaced points. Burgers' equation was solved using a second-order finite difference scheme.

Burgers' equation can be linearised through the Cole–Hopf transformation. Namely, let $u(x, t)$ be a solution to Burgers' equation and define the function:

$$v(x, t) = exp\left[-\frac{\epsilon}{2\mu}\int_0^x u(s, t)ds\right]. \tag{4.2}$$

If $u(x, t)$ satisfies Burgers' equation, then $v(x, t)$ solves the heat equation. The function $u(x, t)$ can be recovered from $v(x, t)$ by inverting the transformation:

$$u = -2\frac{\mu}{\epsilon}\frac{v_x}{v}. \tag{4.3}$$

A general schematic of the neural network architecture used for Burgers' equation is shown in Figure 4. The interpretation of the network in the context of the Cole–Hopf transformation is as follows. The input to the network $\mathbf{u_k}$ is the discretised solution to Burgers' equation at time t_k. The outer encoder $\chi + \mathbf{I}$ performs a linearising transformation analogous to equation (4.2). The inner encoder ψ_e diagonalises the system and potentially reduces the dimensionality of the system. Since the heat equation governs the dynamics of the intrinsic coordinates, ψ_e plays the role of the DFT just like the encoder in the previous section. The matrix $\mathbf{K}$ moves the solution forward in time by one time step. The inner decoder attempts to invert the inner encoder. Finally, the outer decoder $\zeta + \mathbf{I}$ performs a transformation back to the original coordinates, analogous to equation (4.3). The output $\mathbf{u_{k+1}}$ is the solution to Burgers' equation at time $t_{k+1} = t_k + \Delta t$.

Contrary to the schematic in Figure 4, the matrix $\mathbf{K}$ is diagonal and not dense for this particular example. For the inner encoder ψ_e and inner decoder ψ_d, we use a single fully connected linear layer with no bias term. For some results presented below, the layer $\mathbf{v}_k$ has the same width as the input and output layers and hence there is no dimensionality reduction. For other results, the

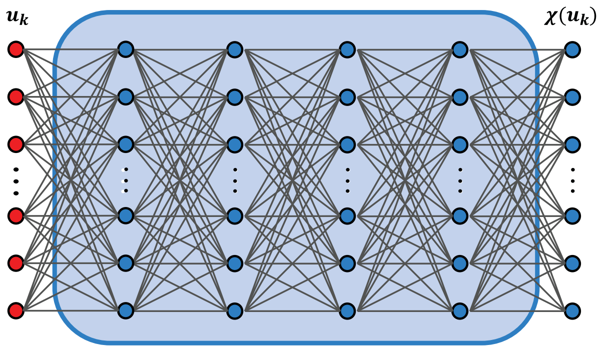

FIGURE 10. The network architecture for the outer encoder and decoder for Burgers' equation. All layers have 128 neurons. ReLU activation functions are used on all but the last layer.

dimensionality is reduced by the inner encoder to some lower rank r. Therefore,

$$\psi_e(\mathbf{u}) = \mathbf{W}_e\mathbf{u}, \qquad \psi_d(\mathbf{u}) = \mathbf{W}_d\mathbf{u}, \tag{4.4}$$

where $\mathbf{W}_e \in \mathbb{R}^{r\times 128}$ and $\mathbf{W}_d \in \mathbb{R}^{128\times r}$. Figure 10 shows the network architecture used for χ in the outer encoder and ζ in the outer decoder. These parts of the network have four fully connected hidden layers that each contain $n = 128$ neurons. All but the output layer use a ReLU activation function. The expression for χ is

$$\chi(\mathbf{u}) = f_5 \circ f_4 \circ f_3 \circ f_2 \circ f_1(\mathbf{u}), \tag{4.5}$$

where $f_j(\mathbf{u}) = \sigma(\mathbf{W}_j\mathbf{u} + \mathbf{b}_j)$ for $j = 1, 2, 3, 4$ and $f_5(\mathbf{u}) = \mathbf{W}_5\mathbf{u} + \mathbf{b}_5$. ζ has the same formula, but with different weights.

4.1 Data

Because the network is not linear, a much more robust data set is necessary in order to have the network discover a transformation that is sufficiently general. We trained neural networks on three different data sets. In all cases, the training data consist of 120,000 trajectories from Burgers' equation, each with 51 equally spaced time steps with $\Delta t = 0.002$. The validation data have the same structure as the training data but with 30,000 trajectories.

The difference in the three training data sets is the diversity of initial conditions. In data set 1, all of the initial conditions are randomly generated white noise. Recall that this was sufficient to find a global transformation for the heat equation. In data set 2, half of the initial conditions are white noise and the other half are sine waves. The sine waves are of the form $u_0(x) = A \sin(\omega x + \phi)$ where $A \in (0, 1)$ and $\phi \in (0, 2\pi)$ are chosen from a latin hypercube sample, and the frequency ω is an integer from 1 to 10 sampled from a truncated geometric distribution. In data set 3, one-third of the trajectories have a white noise initial condition, one-third have a sine wave initial condition and one-third have a square wave initial condition where the height, width and centre of the square wave all come from a latin hypercube sample.

We trained networks on each of the three data sets. For each, we trained a full-width network with no dimensionality reduction in the middle layers, and we also trained a neural network with

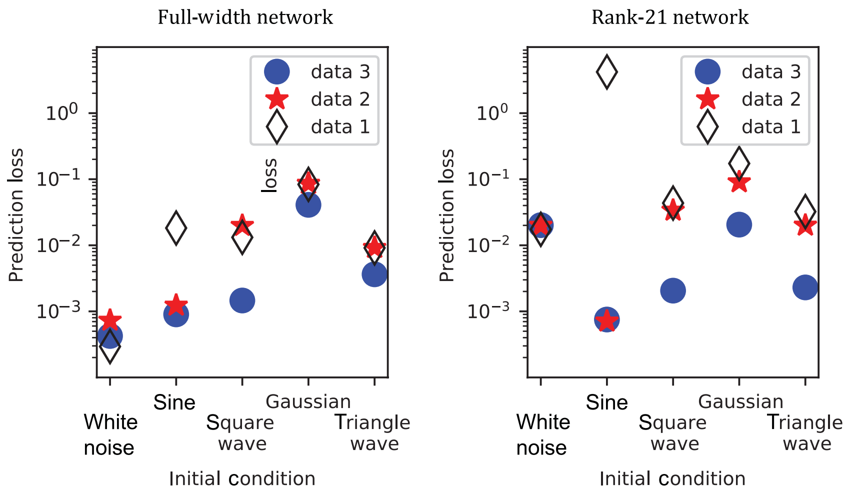

FIGURE 11. The network represented by blue circles was trained on white noise, sine wave and square wave initial conditions. The red star network was trained on white noise and sines, and the black diamond network was just trained on white noise.

rank $r = 21$ in the middle layers. We then tested each network on test data that contain trajectories with five different types of initial conditions. The first three are the same as training data set 3 – white noise, sine wave and square wave. The last two types of initial conditions are functions that were not seen in the training data for any of the data sets – Gaussians and triangle waves. The prediction losses for each network and each type of initial condition are shown in Figure 11. The plot on the left is for full-width networks and the plot on the right is for the reduced rank $r = 21$. In both cases, the network trained on white noise has very low prediction loss for trajectories that have white noise initial conditions but does poorly on other types of initial conditions. Networks trained on white noise and sine waves have low prediction loss for those two types of initial conditions but still perform poorly on square wave initial conditions. The networks trained on white noise, sine waves and square waves not only perform better than the other types of networks on predicting square wave initial conditions but also generalise better to both Gaussians and triangle waves. This demonstrates that a variety of different initial conditions are needed in the training data in order to find a coordinate transformation that can represent any type of function.

Because the networks trained on data set 3 give better global transformations, all of the following results use that training and validation data. Figure 12 shows the prediction given by the full-width network trained on data set 3 for a test trajectory with each of the five initial conditions in the test data. There is very good agreement between the exact solution and the network prediction in each case.

4.2 Comparison with Cole–Hopf

Note that with $\epsilon = 10$ and $\mu = 1$, equation (4.2) is $v(x, t) = e^{-5\int_0^x u(s,t)ds}$. Therefore, even if the Burgers' solution has an ℓ_1 norm that is $\mathcal{O}(1)$, the function v can have values on the order of e^5. This range of scales can be very difficult for a neural network to represent, especially with the use of regularisation. However, similar to the heat equation, the Cole–Hopf transformation

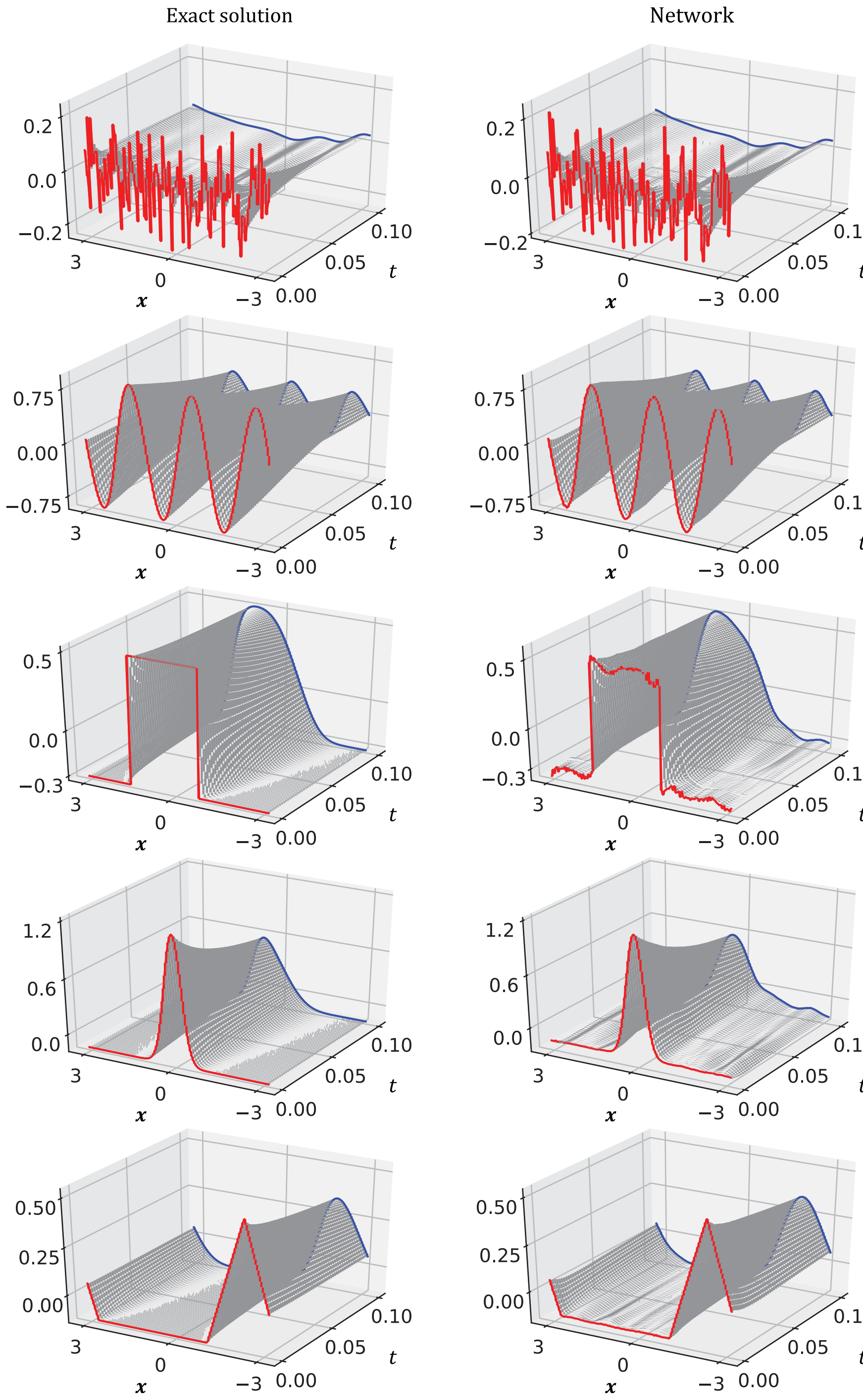

FIGURE 12. A comparison of the exact solution to Burgers' equation and the predictions given by the neural network. The top three use initial conditions of the same type as the training data, but the last two are types of initial conditions not found in the training data. For the prediction at the initial time $t = 0$, the initial condition is encoded and then decoded so the red curve gives an indication of the performance of the autoencoder.

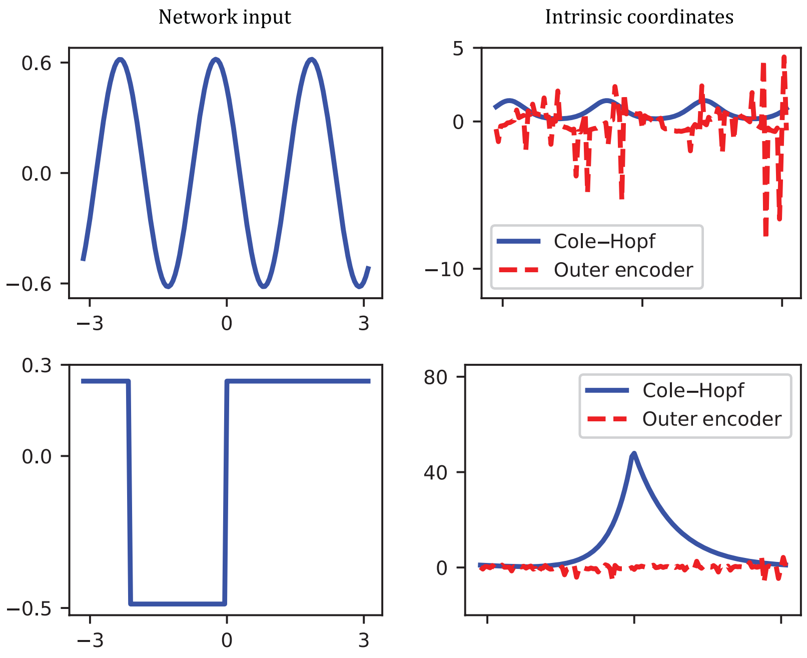

FIGURE 13. A comparison of the Cole–Hopf transformation given by equation (4.2) and the output of the outer encoder of the neural network.

is not unique and can include scaling as long as that scaling is accounted for in the inverse transformation. Therefore, we do not expect the network to reproduce the formulas given by equations (4.2) and (4.3). Indeed, this is true. Figure 13 shows a comparison between the Cole–Hopf transformation and the 'partially encoded' function obtained by feeding the input through the network's outer encoder $\chi + \mathbf{I}$. The plots on the left show two functions $u(x,t)$. The plots on the right show the function $v(x,t)$ given by equation (4.2) as a solid (blue) line and the output of the outer encoder as a dashed red line. The differences are stark. Notice for the square wave on the bottom that the Cole–Hopf transformation gives a function that is large in magnitude, while the network outputs something much smaller in magnitude. However, the linear dynamics in this coordinate system still give good prediction for the Burgers' equation, as evidenced by Figure 12.

4.3 Reduced-order model

By adjusting the widths of the middle layers of the network (labelled $\mathbf{v_k}$ and $\mathbf{v_{k+1}}$ in Figure 4), we can control the rank of the reduced-order model of Burgers' equation. Recall that $r = 128$ is the full-order model. We trained several networks with rank ranging from $r = 1$ to $r = 128$. Figure 14 shows the total validation loss for each network. As expected, the loss decreases as the rank is increased.

4.4 Poorly selected data or architecture

When training a neural network for dynamics, it is important to use sufficiently diverse data as well as the proper architecture. Figure 15 shows the network predictions in the following cases. The top right plot shows the prediction from a neural network that is trained with the architecture

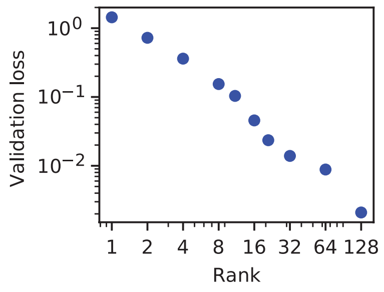

FIGURE 14. A plot of the validation losses for networks of different ranks.

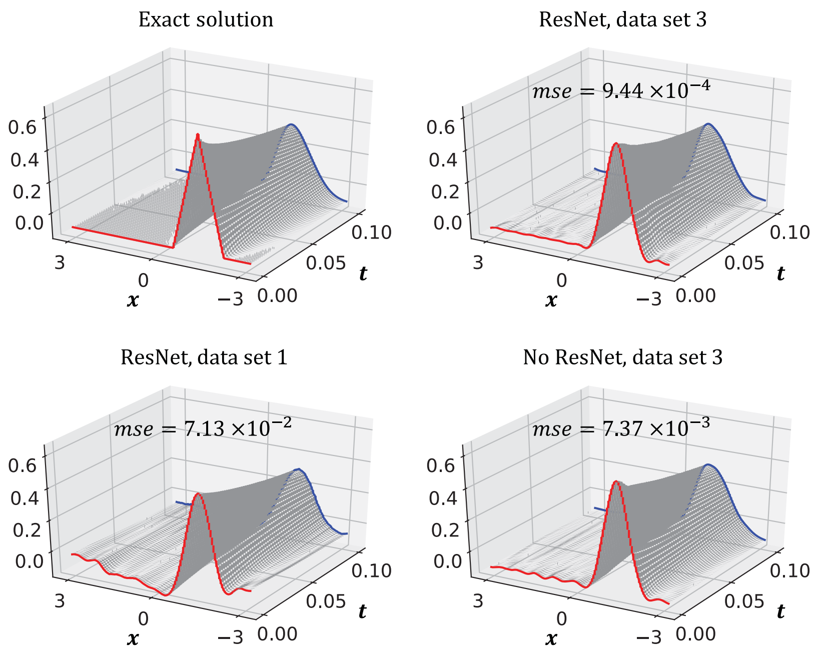

FIGURE 15. A comparison of the network predictions when the data are not sufficiently diverse, the architecture is not chosen properly, and when both the data and architecture are good.

given by Figure 4 and data set 3. This gives the best result because the data have a diverse set of initial conditions and the architecture properly handles the identity. The bottom-left plot shows the same architecture but trained on data set 1, which is less diverse than data set 3. The bottom-right plot shows the prediction from a neural network with the same architecture as Figure 4 but without the skip connections that add the identity. All cases use a reduced-order model with $r = 21$.

5 KS equation

Our final example is the KS equation:

$$u_t = -uu_x - u_{xx} - u_{xxxx}, \qquad x \in (-4\pi, 4\pi). \tag{5.1}$$

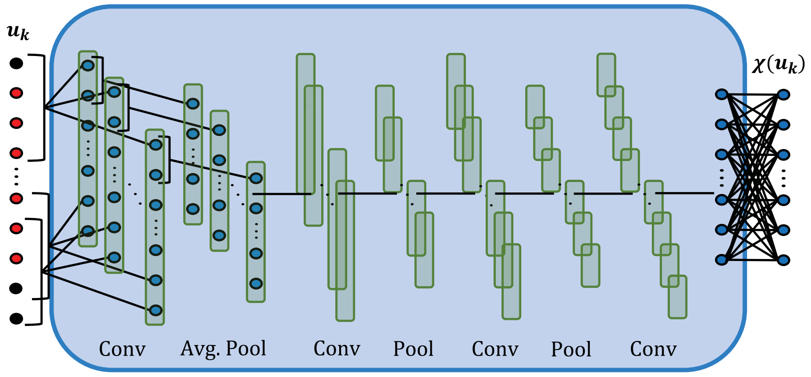

FIGURE 16. The network architecture for the outer encoder and decoder for the KS equation.

As in the previous examples, we use periodic boundary conditions and a spatial discretisation of 128 equally spaced points. The equation was solved in Fourier space using ETDRK4 time-stepping [18]. Unlike Burgers' equation, there is no known coordinate transformation to linearise the KS equation. Therefore, the networks presented below provide the first known invertible transformation to linearise the KS equation.

For the KS equation, we use a different neural network architecture than we used for Burgers' equation in order to demonstrate that our overall methodology is flexible and can accept different architectures for the encoder and decoder. There are two differences between the network architecture for the KS equation and the previous architecture used for Burgers' equation. The matrix $\mathbf{K}$ is now dense instead of being diagonal. Also, the architecture of the outer encoder and decoder is different. In these parts of the network, we use one-dimensional convolutional layers. Convolutional layers typically consist of multiple filters and feature maps. Let $\mathbf{u} \in \mathbb{R}^n$ indexed by $j = 0, \ldots, n-1$ be an input to a convolutional layer and $\mathbf{w} \in \mathbb{R}^m$ indexed by $j = 0, \ldots, m-1$ be a filter. Then, the feature map $\mathbf{s}$ corresponding to the filter $\mathbf{w}$ is given by

$$\mathbf{s}_i = (\mathbf{u} * \mathbf{w})_i = \sum_j \mathbf{u}_j \mathbf{w}_{i-j}, \tag{5.2}$$

where the summation is over all j such that the indices are defined. The architecture used for the KS equation is shown in Figure 16. The first hidden layer is a convolutional layer containing eight filters followed by an average pooling layer. This is then followed by a convolutional layer with 16 filters and an average pooling layer, a convolutional layer with 32 filters followed by an average pooling layer, and finally a convolutional layer with 64 filters. In all cases, the convolutional layers have kernel size 4, stride length 1, zero-padding and ReLU activation functions, while the average pooling layers have pool size 2, stride length 2 and no padding. The last convolutional layer is followed by a fully connected layer with 128 neurons and ReLU activation and then a final fully connected linear layer.

The data set mirrors the data used for Burgers' equation. The training data consist of 120,000 trajectories, each with 51 equally spaced time steps. The initial conditions are evenly split between white noise, sine waves and square waves. The validation data have the same structure as the training data but with 30,000 trajectories. The test data include white noise, sine wave and square wave initial conditions as well as Gaussian and triangle wave initial conditions.

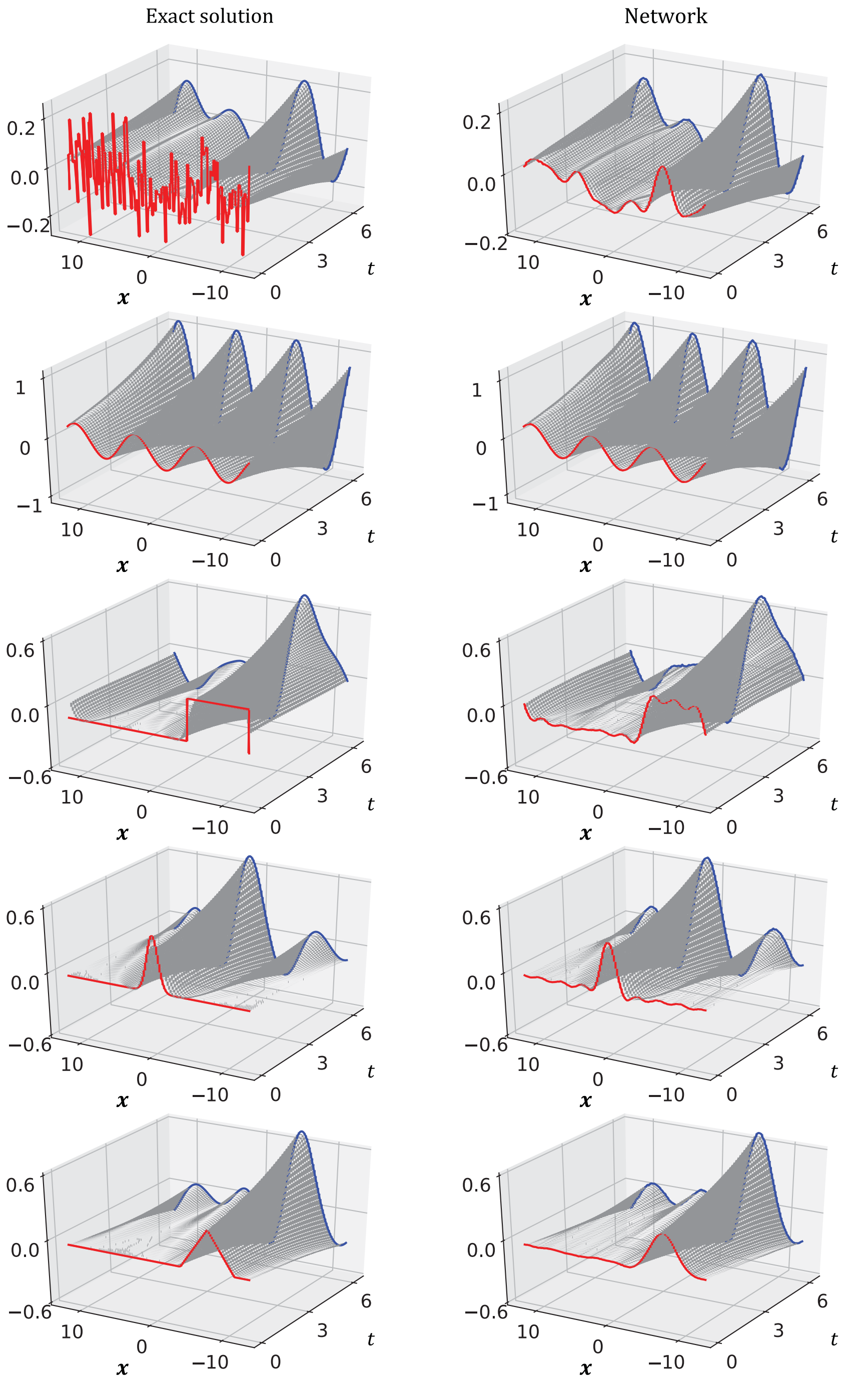

FIGURE 17. A comparison of the exact solution to the KS equation and the predictions given by the neural network. The top three use initial conditions of the same type as the training data, but the last two are types of initial conditions not found in the training data. For the prediction at the initial time $t = 0$, the initial condition is encoded and then decoded so the red curve gives an indication of the performance of the autoencoder.

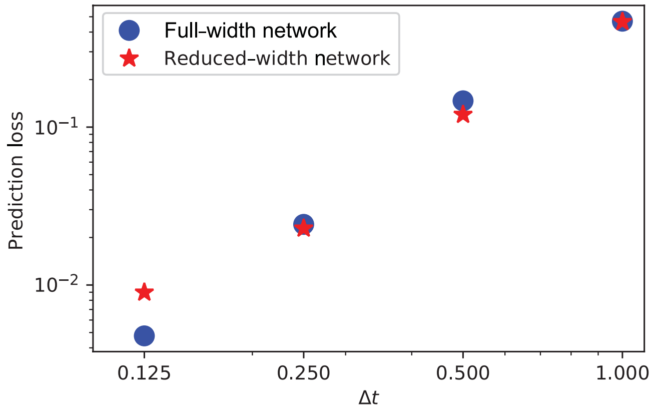

FIGURE 18. Prediction loss for the KS Equation for data with different time steps.

We trained networks for several different data sets with the only difference being the time step. The time steps considered are $\Delta t = 0.125$, $\Delta t = 0.25$, $\Delta t = 0.5$ and $\Delta t = 1$. For each data set, we trained a full-width network (no dimensionality reduction) and a reduced-width network with rank $r = 21$.

Figure 17 shows the predictions from the network trained with $\Delta t = 0.125$ and $r = 21$. The exact solution is shown on the left for five different initial conditions and the network output is on the right. The top three plots use initial conditions of the type used in the training data, so good predictive power is expected. However, the bottom two plots use initial conditions that are not represented in the training data. The good agreement between the exact solution and the network predictions in these two cases show that the network gives a good global transformation to linearise the dynamics.

Figure 18 shows the prediction loss on the test data for networks trained on data with different time steps. The results are given for both full-width as well as reduced-width networks. The prediction loss is much lower for smaller time steps. This behaviour is expected because a smaller time step leads to the solutions at consecutive times being closer together and therefore the network needs to approximate something nearer to the identity (and hence nearer to linear).

Note that the Cole–Hopf transformation does not depend on a time step. Similarly, the coordinate transformation for the KS equation should be independent of the time step, and therefore the encoder and decoder of the neural networks should be the same for data with any time step. The fact that smaller time steps are easier to represent with a neural network leads to a strategy for getting accurate predictions for larger time steps. In order to train the network for the data with $\Delta t = 0.25$, we initialised the weights with the weights from the network that had been trained with the shorter time step $\Delta t = 0.125$. This homotopy from the shorter time step to the longer time step led to both faster training and more accurate results. The total validation loss for $\Delta t = 0.25$ was 0.0192 for the full-width network and 0.0257 for the reduced-width network when using the homotopy strategy. When training the networks from scratch, the total validation losses were 0.0254 for the full-width network and 0.1753 for the reduced-width network. Therefore, failure to use a homotopy from the shorter time step led to a 32% increase in the total validation loss for the full-width network and a 582% increase for the reduced-width network. For $\Delta t = 0.5$, we used a homotopy from the $\Delta t = 0.25$ time step. Failure to homotopy led to a 45% increase in the validation loss for the full-width network and a 25% increase for the reduced-width network.

6 Conclusion

In this work, we have developed a scalable deep learning architecture specifically designed to learn a change of coordinates in which a given partial differential equation becomes linear. In particular, a custom deep autoencoder network is designed with additional constraints that in the low-dimensional latent space the system must evolve linearly in time. The resulting coordinate transformation is able to identify linearising transformations that are analogous to known transformations, such as the Cole-Hopf transformation that maps the non-linear Burgers' equation into the linear heat equation. However, the proposed architecture is designed to generalise to more complex PDE systems, such as the demonstrated KS. Because our network is constrained to learn a linear dynamical system, the resulting network is interpretable, with the encoder network identifying the eigenspace of the Koopman operator.

There are a number of future directions that arise from this work. The analysis performed here is promising and motivates the application of this approach on new PDE models for which known linearising transformations do not exist. Extending these methods to higher-dimensional problems in two and three dimensions would also be interesting. Complex systems in fluid mechanics and turbulence, that exhibit multiscale phenomena in space and time, are especially interesting. These systems are generally characterised by a continuous frequency spectrum, motivating parameterised latent-space dynamics or time delays, which have both been successful for modelling ODEs with continuous spectra. Incorporating additional invariants and symmetries in the network architecture is another promising avenue of future work, as there is evidence that encoding partially known physics improves the learning rates and generalisability of the resulting models. The ability to embed non-linear systems in a linear framework is particularly useful for estimation and control, where a wealth of techniques exist for linear systems. Therefore, it will likely be fruitful to extend these approaches to include inputs and control.

Acknowledgements

We would first like to thank Kathleen Champion, Jean-Christophe Loiseau, Karthik Duraisamy, Frank Noe, Yannis Kevrekidis and Igor Mezic for valuable discussions about sparse dynamical systems and autoencoders. The authors acknowledge support from the Defense Advanced Research Projects Agency (DARPA PA-18-01-FP-125). SLB further acknowledges support from the Army Research Office (ARO W911NF-17-1-0306), and JNK from the Air Force Office of Scientific Research (FA9550-17-1-0329). This work was facilitated through the use of advanced computational, storage and networking infrastructure provided by AWS cloud computing credits funded by the STF at the University of Washington. This research was funded in part by the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357.

Conflict of interest

The authors have no conflict of interest.

References

[1] Ablowitz, M. J. & Segur, H. (1981) *Solitons and the Inverse Scattering Transform*, Vol. 4, SIAM, Philadelphia, PA.

[2] Bishop, C. (2006) *Pattern Recognition and Machine Learning*, Springer, New York, NY.

[3] Boyce, W. E. & DiPrima, R. C. (2008) *Elementary Differential Equations*, 9th ed., Wiley, Hoboken, NJ.

[4] Brunton, S. L., Brunton, B. W., Proctor, J. L. & Kutz, J. N. (2016) Koopman invariant subspaces and finite linear representations of nonlinear dynamical systems for control. *PLOS ONE* **11**, 1–19.

[5] Brunton, S. L. & Kutz, J. N. (2019) *Data-Driven Science and Engineering: Machine Learning, Dynamical Systems, and Control*, Cambridge University Press, Cambridge.

[6] Budišić, M. & Mezić, I. (2012) Geometry of the ergodic quotient reveals coherent structures in flows. *Physica D Nonlinear Phenomena* **241**, 1255–1269.

[7] Champion, K., Lusch, B., Kutz, J. N. & Brunton, S. L. (2019) Data-driven discovery of coordinates and governing equations. *Proc. Nat. Acad. Sci.* **116**, 22445–22451.

[8] Cole, J. D. (1951) On a quasi-linear parabolic equation occurring in aerodynamics. *Quart. Appl. Math.* **9**, 225–236.

[9] Cybenko, G. (1989) Approximation by superpositions of a sigmoidal function. *Math. Control Sig. Syst. (MCSS)* **2**, 303–314.

[10] Dsilva, C. J., Talmon, R., Coifman, R. R. & Kevrekidis, I. G. (2018) Parsimonious representation of nonlinear dynamical systems through manifold learning: a chemotaxis case study. *Appl. Comput. Harmonic Anal.* **44**, 759–773.

[11] Foias, C., Jolly, M., Kevrekidis, I. & Titi, E. (1994) On some dissipative fully discrete nonlinear Galerkin schemes for the Kuramoto-Sivashinsky equation. *Phys. Lett. A* **186**, 87–96.

[12] Gonzalez-Garcia, R., Rico-Martinez, R. & Kevrekidis, I. (1998) Identification of distributed parameter systems: a neural net based approach. *Comput. Chem. Eng.* **22**, S965–S968.

[13] Goodfellow, I., Bengio, Y. & Courville, A. (2016) *Deep Learning*, MIT Press, Cambridge, MA. http://www.deeplearningbook.org.

[14] Haberman, R. (2004) *Applied Partial Differential Equations: with Fourier Series and Boundary Value Problems*, Pearson Prentice Hall, Upper Saddle River, NJ.

[15] He, K., Zhang, X., Ren, S. & Sun, J. (2016) Deep residual learning for image recognition. In: *Proceedings of the IEEE Conference on Ccomputer Vision and Pattern Recognition*, pp. 770–778.

[16] Hopf, E. (1950) The partial differential equation $u_t + uu_x = \mu u_{xx}$. *Comm. Pure App. Math.* **3**, 201–230.

[17] Hornik, K., Stinchcombe, M. & White, H. (1990) Universal approximation of an unknown mapping and its derivatives using multilayer feedforward networks. *Neural Networks* **3**, 551–560.

[18] Kassam, A. & Trefethen, L. (2005) Fourth-order time stepping for stiff PDEs. *SIAM J. Sci. Comput.* **26**, 1214–1233.

[19] Klus, S., Nüske, F., Koltai, P., Wu, H., Kevrekidis, I., Schütte, C. & Noé, F. (2018) Data-driven model reduction and transfer operator approximation. *J. Nonlinear Sci.* **28**, 985–1010.

[20] Koopman, B. O. (1931) Hamiltonian systems and transformation in Hilbert space. *Proc. Nat. Acad. Sci.* **17**, 315–318.

[21] Kutz, J. N. (2013) *Data-Driven Modeling & Scientific Computation: Methods for Complex Systems & Big Data*, Oxford University Press, Oxford.

[22] Kutz, J. N., Brunton, S. L., Brunton, B. W. & Proctor, J. L. (2016) *Dynamic Mode Decomposition: Data-Driven Modeling of Complex Systems*, SIAM, Philadelphia, PA.

[23] Kutz, J. N., Proctor, J. L. & Brunton, S. L. (2018) Applied Koopman theory for partial differential equations and data-driven modeling of spatio-temporal systems. *Complexity* **2018**, 1–16.

[24] Li, Q., Dietrich, F., Bollt, E. M. & Kevrekidis, I. G. (2017) Extended dynamic mode decomposition with dictionary learning: a data-driven adaptive spectral decomposition of the Koopman operator. *Chaos Interdiscip. J. Nonlinear Sci.* **27**, 103111.

[25] Lu, L., Shin, S., Su, Y. & Karniadakis, G. (2019) Dying ReLU and initialization: theory and numerical examples. arXiv:1903.06733.

[26] Lu, L., Su, Y. & Karniadakis, G. (2018) Collapse of deep and narrow neural nets. arXiv:1808.04947.

[27] Lusch, B., Kutz, J. N. & Brunton, S. L. (2018) Deep learning for universal linear embeddings of nonlinear dynamics. *Nat. Commun.* **9**, 4950.

[28] Mallat, S. (2016) Understanding deep convolutional networks. *Philos. Trans. R. Soc. A Math. Phys. Eng. Sci.* **374**, 20150203.

[29] Mardt, A., Pasquali, L., Wu, H. & Noé, F. (2018) VAMPnets: deep learning of molecular kinetics. *Nat. Commun.* **9**, 5.

[30] Mezić, I. (2005) Spectral properties of dynamical systems, model reduction and decompositions. *Nonlinear Dyn.* **41**, 309–325.

[31] Mezić, I. (2013) Analysis of fluid flows via spectral properties of the Koopman operator. *Ann. Rev. Fluid Mech.* **45**, 357–378.

[32] Mezić, I. & Banaszuk, A. (2004) Comparison of systems with complex behavior. *Physica D Nonlinear Phenomena* **197**, 101–133.

[33] Neu, J. C. (1980) The method of near-identity transformations and its applications. *SIAM J. Appl. Math.* **38**, 189–208.

[34] Noé, F. & Nüske, F. (2013) A variational approach to modeling slow processes in stochastic dynamical systems. *Multiscale Model. Simul.* **11**, 635–655.

[35] Nüske, F., Keller, B. G., Pérez-Hernández, G., Mey, A. S. & Noé, F. (2014) Variational approach to molecular kinetics. *J. Chem. Theory Comput.* **10**, 1739–1752.

[36] Otto, S. E. & Rowley, C. W. (2019) Linearly-recurrent autoencoder networks for learning dynamics. *SIAM J. Appl. Dyn. Syst.* **18**, 558–593.

[37] Page, J. & Kerswell, R. R. (2018) Koopman analysis of burgers equation. *Phys. Rev. Fluids* **3**, 071901.

[38] Pan, S. & Duraisamy, K. (2020) Physics-informed probabilistic learning of linear embeddings of nonlinear dynamics with guaranteed stability. *SIAM J. Appl. Dyn. Syst.* **19**, 480–509.

[39] Rico-Martinez, R., Kevrekidis, I. & Krischer, K. (1995) Nonlinear system identification using neural networks: dynamics and instabilities. In: *Neural Networks for Chemical Engineers*, pp. 409–442.

[40] Rowley, C. W., Mezić, I., Bagheri, S., Schlatter, P. & Henningson, D. (2009) Spectral analysis of nonlinear flows. *J. Fluid Mech.* **645**, 115–127.

[41] Schmid, P. J. (2010) Dynamic mode decomposition of numerical and experimental data. *J. Fluid Mech.* **656**, 5–28.

[42] Takeishi, N., Kawahara, Y. & Yairi, T. (2017) Learning Koopman invariant subspaces for dynamic mode decomposition. In: *Advances in Neural Information Processing Systems*, pp. 1130–1140.

[43] Wehmeyer, C. & Noé, F. (2017) Time-lagged autoencoders: deep learning of slow collective variables for molecular kinetics. *J. Chem. Phys.* **148**, 241703.

[44] Wiggins, S. (2003) *Introduction to Applied Nonlinear Dynamical Systems and Chaos*, Vol. 2, Springer, New York, NY.

[45] Williams, M. O., Kevrekidis, I. G. & Rowley, C. W. (2015) A data-driven approximation of the Koopman operator: extending dynamic mode decomposition. *J. Nonlinear Sci.* **25**, 1307–1346.

[46] Williams, M. O., Rowley, C. W. & Kevrekidis, I. G. (2015) A kernel-based method for data-driven Koopman spectral analysis. *J. Comput. Dyn.* **2**, 247–265.

[47] Yeung, E., Kundu, S. & Hodas, N. (2019) Learning deep neural network representations for Koopman operators of nonlinear dynamical systems. In: *2019 American Control Conference (ACC)*, pp. 4832–4839.

Appendix A. Neural network training

All of the code and results for this project can be found at https://github.com/CraigGin/PDEKoopman. All of the neural networks were trained using Tensorflow. The networks have real-valued weights and are initialised in one of two ways. The layers in the outer encoder and outer decoder are initialised using He initialisation. The **K** matrix is initialised as the identity matrix. For the inner encoder and inner decoder, they are initialised as the identity matrix when they are square matrices. For non-square matrices, they are initialised as 'identity-like' matrices. In particular, for an $m \times n$ matrix with $n > m$, each row has $n - m + 1$ non-zero entries that are all $1/(n - m + 1)$. For row j, the non-zero entries are in columns j to $j + n - m$. So, for example, the initialisation for a 3×5 matrix is

$$\begin{pmatrix} \frac{1}{3} & \frac{1}{3} & \frac{1}{3} & 0 & 0 \\ 0 & \frac{1}{3} & \frac{1}{3} & \frac{1}{3} & 0 \\ 0 & 0 & \frac{1}{3} & \frac{1}{3} & \frac{1}{3} \end{pmatrix}.$$

If $m > n$, then the matrix is initialised as the transpose of what is described above. Note that if $n = m$, this is the identity matrix.

ReLU activation functions are used on all hidden layers of the outer encoder and outer decoder. All other layers use no activation function. The inner encoder and decoder and the **K** matrix do not have a bias term.

We used the Adam optimiser to train the networks with a random learning rate between 10^{-6} and 10^{-3}. The batch size was 64. ℓ^2 regularisation was used on all weight matrices except for **K** and with a regularisation factor of 10^{-8}.

For each experiment, 20 neural networks were trained with a random learning rate and random initialisation (for layers initialised with He initialisation). Each was trained for 20 min. Then, the network with the lowest validation loss was loaded and trained until convergence.