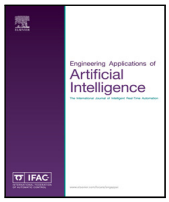
Engineering Applications of Artificial Intelligence 96 (2020) 103996

Contents lists available at ScienceDirect

Engineering Applications of Artificial Intelligence

journal homepage: www.elsevier.com/locate/engappai



A tutorial on solving ordinary differential equations using Python and hybrid physics-informed neural network

Renato G. Nascimento, Kajetan Fricke, Felipe A.C. Viana *

Department of Mechanical and Aerospace Engineering, University of Central Florida, Orlando, FL 32816-8030, USA

ARTICLE INFO

Keywords:
Physics-informed neural network
Scientific machine learning
Uncertainty quantification
Hybrid model python implementation

ABSTRACT

We present a tutorial on how to directly implement integration of ordinary differential equations through recurrent neural networks using Python. In order to simplify the implementation, we leveraged modern machine learning frameworks such as TensorFlow and Keras. Besides, offering implementation of basic models (such as multilayer perceptrons and recurrent neural networks) and optimization methods, these frameworks offer powerful automatic differentiation. With all that, the main advantage of our approach is that one can implement hybrid models combining physics-informed and data-driven kernels, where data-driven kernels are used to reduce the gap between predictions and observations. Alternatively, we can also perform model parameter identification. In order to illustrate our approach, we used two case studies. The first one consisted of performing fatigue crack growth integration through Euler's forward method using a hybrid model combining a data-driven stress intensity range model with a physics-based crack length increment model. The second case study consisted of performing model parameter identification of a dynamic two-degree-of-freedom system through Runge–Kutta integration. The examples presented here as well as source codes are all open-source under the GitHub repository https://github.com/PML-UCF/pinn_code_tutorial.

1. Introduction

Deep learning and physics-informed neural networks (Cheng et al., 2018; Shen et al., 2018; Chen et al., 2018; Pang and Karniadakis, 2020) have received growing attention in science and engineering over the past few years. The fundamental idea, particularly with physics-informed neural networks, is to leverage laws of physics in the form of differential equations in the training of neural networks. This is fundamentally different than using neural networks as surrogate models trained with data collected at a combination of inputs and output values. Physics-informed neural networks can be used to solve the forward problem (estimation of response) and/or the inverse problem (model parameter identification).

Although there is no consensus on nomenclature or formulation, we see two different and very broad approaches to physics-informed neural network. There are those using neural network as approximate solutions for the differential equations (Chen et al., 2018; Raissi et al., 2019; Raissi and Karniadakis, 2018; Pan and Duraisamy, 2020). Essentially, through collocation points, the neural network hyperparameters are optimized to satisfy initial/boundary conditions as well as the constitutive differential equation itself. For example, Raissi et al. (2019) present an approach for solving and discovering the form of differential equations using neural networks, which has a companion GitHub repository (https://github.com/maziarraissi/PINNs) with detailed and documented Python implementation. Authors proposed using deep neural networks to handle the direct problem of solving differential equations through the loss function (functional used in the optimization of hyperparameters). The formulation is such that neural networks are parametric trial solutions of the differential equation and the loss function accounts for errors with respect to initial/boundary conditions and collocation points. Authors also present a formulation for learning the coefficients of differential equations given observed data (i.e., calibration). The proposed method is applied to both, the Schroedinger equation, a partial differential equation utilized in quantum mechanics systems, and the Allen–Cahn equation, an established equation for describing reaction–diffusion systems. Pan and Duraisamy (2020) introduced a physics informed machine learning approach to learn the continuous-time Koopman operator. Authors apply the derived method to nonlinear dynamical systems, in particular within the field of fluid dynamics, such as modeling the unstable wake flow behind a cylinder. In order to derive the method, authors used a measure-theoretic approach to create a deep neural network. Both differential and recurrent model types are derived, where the latter is used when discrete trajectory data can be obtained whereas the

* Corresponding author.
E-mail address: viana@ucf.edu (F.A.C. Viana).
URL: https://pml-ucf.github.io/ (F.A.C. Viana).

https://doi.org/10.1016/j.engappai.2020.103996
Received 18 June 2020; Received in revised form 15 September 2020; Accepted 2 October 2020
Available online xxxx
0952-1976/

differential form is suitable when governing equations are disposable. This physics-informed neural network approach shows its strength regarding uncertainty quantification and is robust against noisy input signal.

Alternatively, there are those building hybrid models that directly code reduced order physics-informed models within deep neural networks (Nascimento and Viana, 2020; Yucesan and Viana, 2020; Dourado and Viana, 2020; Karpatne et al., 2017; Singh et al., 2019). This implies that the computational cost of these physics-informed kernels have to be comparable to the linear algebra found in neural network architectures. It also means that tuning of the physics-informed kernel hyperparameters through backpropagation requires that adjoints to be readily available (through automatic differentiation (Baydin et al., 2018), for example). For example, Yucesan and Viana (2020) proposed a hybrid modeling approach which combines reduced-order models and machine learning for improving the accuracy of cumulative damage models used to predict wind turbine main bearing fatigue. The reduce-order models capture the behavior of bearing loads and bearing fatigue; while machine learning models account for uncertainty in grease degradation. The model was successfully used to predict grease degradation and bearing fatigue across a wind park; and with that, optimize the regreasing intervals. Karpatne et al. (2017) presented an interesting taxonomy for what authors called theory-guided data science. In the paper, they discuss how one could augment machine learning models with physics-based domain knowledge and walk from simple correlation-based models, to hybrid models, to fully physics-informed machine learning (such as in solving differential equations directly). Authors discuss examples in hydrological modeling, computational chemistry, mapping surface water dynamics, and turbulence modeling.

We will focus on discussing a Python implementation for hybrid physics-informed neural networks. We believe these hybrid implementations can have an impact in real-life applications, where reduced order models capturing the physics are available and well adopted. Most of the time, the computational efficiency of reduced order models comes at the cost of loss of physical fidelity. Hybrid implementations of physics-informed neural networks can help reducing the gap between predictions and observed data.

Our approach starts with the analytical formulation and passes through the numerical integration method before landing in the neural network implementation. Depending on the application, different numerical integration methods can be used. While this is an interesting topic, it is not the focus of our paper. Instead, **we will focus on how to move from analytical formulation to numerical implementation of the physics-informed neural network model.**

We will address the implementation of ordinary differential equation solvers using two case studies in engineering. Fatigue crack propagation is used as an example of first order ordinary differential equations. In this example, **we show how physics-informed neural networks can be used to mitigate epistemic (model-form) uncertainty in reduced order models.** Forced vibration of a 2 degree-of-freedom system is used as an example of a system of second order ordinary differential equations. In this example, **we show how physics-informed neural networks can be used to estimate model parameters of a physical system.** The main intend of this paper is to be a tutorial for a hybrid implementation of physics-informed neural networks. The remaining of the paper is organized as follows. Section 2 specifies the implementation choices in terms of language and libraries, and public repositories (needed for replication of results). Section 3.2 presents the formulation and implementation for integrating first order ordinary differential equation with the simple Euler's forward method. Section 3.3 details the formulation and implementation for integrating a system of coupled second order differential equations with the Runge–Kutta method. Section 4 closes the paper recapitulating salient points and presenting conclusions and future work. Finally, Appendix summarizes concepts about neural networks used in this paper.

2. Code repository and replication of results

In this paper, we will use TensorFlow (Abadi et al., 2016) (version 2.0.0-beta1), Keras (Chollet et al., 2015), and the Python application programming interface. We will leverage the object orientation capabilities of the framework to differentiate classes that will implement the Euler's forward method. Further information on how to customize neural network architectures within TensorFlow, the reader is referred to the TensorFlow documentation (tensorflow.org).

In order to replicate our results, the interested reader can download codes and data available at Fricke et al. (2020). Throughout this paper, we will highlight the main features of the codes found in this repository. We also refer to the PINN package (Viana et al., 2019) (a freely available base package for physics-informed neural network, which contains specialized implementations and examples of cumulative damage models).

3. Physics-informed neural network for ordinary differential equations

In this section, we will focus on our hybrid physics-informed neural network implementation for ordinary differential equations. This is specially useful for problems where physics-informed models are available, but known to have predictive limitations due to model-form uncertainty or model-parameter uncertainty. We start by providing the background on recurrent neural networks and then discuss how we implement them for numerical integration.

3.1. Background: Recurrent neural networks

Recurrent neural networks (Goodfellow et al., 2016) extend traditional feed forward networks to handle time-dependent responses. As illustrated in Fig. 1, in every time step t, recurrent neural networks apply a transformation to a state $\mathbf{y}$ such that:

$$\mathbf{y}_t = f(\mathbf{y}_{t-1}, \mathbf{x}_t), \tag{1}$$

where $t \in [0, \dots, T]$ represent the time discretization; $\mathbf{y} \in \mathbb{R}^{n_y}$ are the states representing the quantities of interest, $\mathbf{x} \in \mathbb{R}^{n_x}$ are input variables; and $f(.)$ is the transformation cell. Depending on the application, $\mathbf{y}$ can be observed in every time step t or only at specific observation times.

Popular recurrent neural network cell designs include the long–short term memory (Hochreiter and Schmidhuber, 1997) and the gated recurrent unit (Cho et al., 2014), as illustrated in Fig. 1. Although very useful in data-driven applications (time-series data Connor et al., 1994; Sak et al., 2014, speech recognition Graves et al., 2013, text sequence Sutskever et al., 2011, etc.), these cell designs do not implement numerical integration directly. In this paper, we will show how to implement specialized recurrent neural networks for numerical integration. The only requirements are that computations stay within linear algebra complexity (so that computational cost stays comparable to any other neural network architecture) and gradients with respect to trainable parameters are made available (so that backpropagation can be used for optimization). Keeping these two constraints in mind, we can design customized recurrent neural network cells that performs the desired integration technique. For the sake of illustration, in Sections 3.2 and 3.3 , we customized two recurrent neural network cells, one for Euler integration and one for Runge–Kutta integration, as shown in Fig. 1.

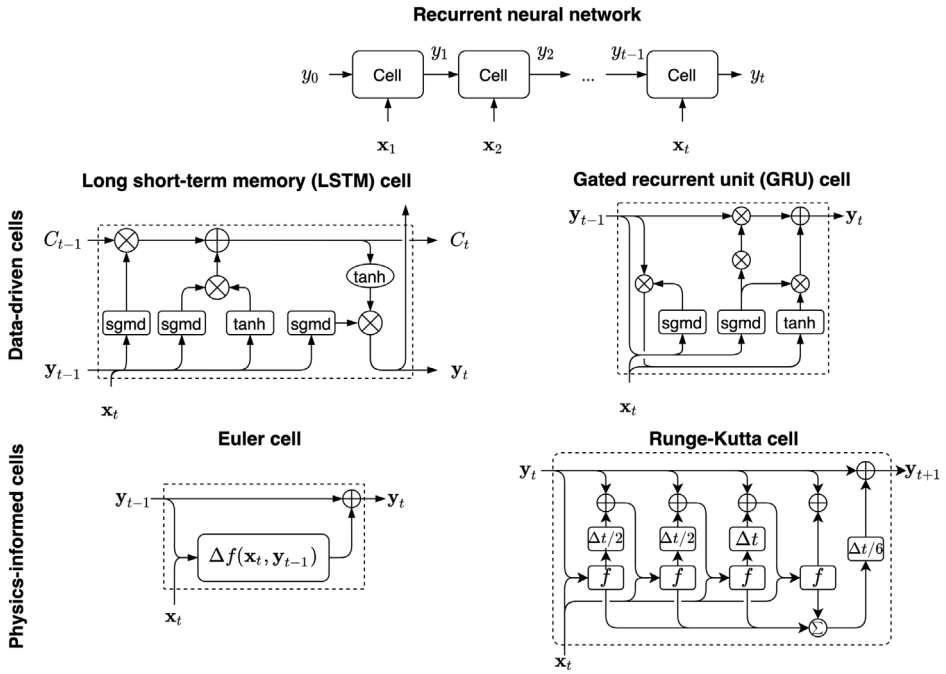


Fig. 1. Recurrent neural network. Two popular cell designs used in data-driven applications are illustrated in contrast with the two physics-informed cells we discuss in this paper.

3.2. First order ordinary differential equations

Consider the first order ordinary differential equation expressed in the form

$$\frac{dy}{dt} = f(\mathbf{x}(t), y, t) \tag{2}$$

where $\mathbf{x}(t)$ are controllable inputs to the system, y is the output of interest, and t is time. The solution to Eq. (2) depends on initial conditions (y at $t = 0$), input data ($\mathbf{x}(t)$ known at different time steps), and the computational cost associated with the evaluation of $f(.)$.

3.2.1. Case study: Fatigue crack propagation

In this case study, we consider the tracking of low cycle fatigue damage. We are particularly interested in a control point that is monitored for a fleet of assets (e.g., compressors, aircraft, etc.). This control point sits on the center of large plate in which loads are applied perpendicularly to the crack plane. As depicted in Fig. 2(a), under such circumstances, fatigue crack growth progresses following Paris law (Paris and Erdogan, 1963)

$$\frac{da}{dN} = C(\Delta K(t))^m \quad \text{and} \quad \Delta K(t) = F\Delta S(t)\sqrt{\pi a(t)}, \tag{3}$$

where a is the fatigue crack length, C and m are material properties, ΔK is the stress intensity range, ΔS is the far-field cyclic stress time history, and F is a dimensionless function of geometry (Dowling, 2012).

We assume that the control point inspection occurs in regular intervals. Scheduled inspection of part of the fleet is adopted to reduce cost associated with it (mainly downtime, parts, and labor). As inspection data is gathered, the predictive models for fatigue damage are updated. In turn, the updated models can be used to guide the decision of which machines should be inspected next.

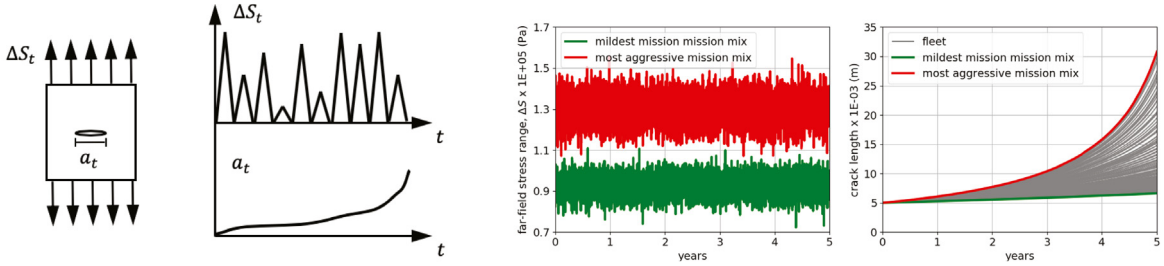
Fig. 2(b) illustrates all the data used in this case study. There are 300 machines, each one accumulating 7300 loading cycles. Not all machines are subjected to the same mission mix. In fact, the duty cycles can greatly vary, driving different fatigue damage accumulation rates throughout the fleet. In this case study, we consider that while the history of cyclic loads is known throughout the operation of the fleet, crack length history is not available. We divided the entire data set consisting of 300 machines into 60 assets used for training and 240 assets providing the test data sets. In real life, these numbers depend on the cost associated with inspection (grounding the aircraft implies in loss of revenue besides cost of the actual inspection). For the sake of this example, we observed 60 time histories of 7300 data points each (total of 438,000 input points) and only 60 output observations. The test data consists of 240 time histories of 7300 data points each (total of 1,752,000 input points) and no crack length observations. In order to highlight the benefits of the hybrid implementation, we use only 60 crack length observations after the entire load cycle regime. The fact that we directly implemented the governing equation in a recurrent neural network cell compensates for the number of points of available output data. Hence, for training procedures we only use the aforementioned 60 assets, while the data for the remaining 240 machines can be utilized as validation data set.

3.2.2. Computational implementation

For the sake of this example, assume that the material is characterized by $C = 1.5 \times 10^{-11}$ and $m = 3.8$, $F = 1$, and that the initial crack length is $a_0 = 0.005$ m. $\Delta S(t)$ is estimated either through structural health monitoring systems or high-fidelity finite element analysis together with cycle count methods (e.g., the rain flow method Collins, 1993). This way, **the numerical method used to solve Eq. (3) hinges on the availability and computational cost associated with $\Delta S(t)$. In this example, let us assume that the far-field stresses are available in every cycle at very low computational cost (for example, the load cases are known and stress analysis is performed before hand).**

Within folder `first_order_ode` of the repository available at Fricke et al. (2020), the interested reader will find all data files used in this case study. File `a0.csv` has the value for the initial crack length ($a_0 = 0.005$ m) used throughout this case study. Files `Stest.csv` and `Strain.csv` contain the load histories for the fleet of 300 machines as well as the 60 machines used in the training of the physics-informed neural network. Files `atest.csv` and `atrain.csv` contain the fatigue crack length histories for the fleet as well as the 60 observed crack lengths used in the training of the physics-informed neural network.

We will then show how ΔK_t can be estimated through a multilayer perceptron (MLP), which works as a corrector on any poor estimation of



(a) Large plate with loads perpendicular to crack plane

(b) Snapshot of fleet-wide data (300 machines)

Fig. 2. Fatigue crack propagation details. Crack growth is governed by Paris law, Eq. (3), a first order ordinary differential equation. The input is time history of far-field stress cyclic loads, ΔS, and the output is the fatigue crack length, a. In this case study, 300 aircraft are submitted to a wide range of loads (due to different missions and mission mixes). This explains the large variability in observed crack length after 5 years of operation.

either $\Delta S(t)$ or ΔK_t (should it have been implement through a physics-informed model). Therefore, we can simply use the Euler's forward method (Press et al., 2007) (with unit time step) to obtain

$$a_n = a_0 + \sum_{t=1}^{n} \Delta a(\Delta S_t, a_{t-1}), \tag{4}$$

$$\Delta a_t = C\Delta K_t^m, \quad \text{and} \quad \Delta K_t = \text{MLP}\left(\Delta S_t, a_{t-1}; \mathbf{w}, \mathbf{b}\right), \tag{5}$$

where $\mathbf{w}$ and $\mathbf{b}$ are the trainable hyperparameters.

Similarly to regular neural networks, we use observed data to tune $\mathbf{w}$ and $\mathbf{b}$ by minimizing a loss function. Here we use the mean squared error:

$$\Lambda = \frac{1}{n}(\mathbf{a} - \hat{\mathbf{a}})^T(\mathbf{a} - \hat{\mathbf{a}}), \tag{6}$$

where n is the number of observations, $\mathbf{a}$ are fatigue crack length observations, and $\hat{\mathbf{a}}$ are the predicted fatigue crack length using the hybrid physics-informed neural network.

Listing 1 lists all the necessary packages. Besides `pandas` and `numpy` for data importation and manipulation, we import a series of packages out of TensorFlow. We import `Dense` and `RNN` to leverage native multilayer perceptron and recurrent neural networks (see Appendix for a brief overview of these architectures). We import `Sequential` and `Layer` so that we can specialize our model. We import `RMSprop` for hyperparameter optimization. Finally, the other operators are needed to move data to TensorFlow-friendly structures.

Listing 2 shows the important snippets of implementation of the Euler integrator cell (to avoid clutter, we leave out the lines that are needed for data-type reinforcement). This is a class inherited from `Layers`, which is what TensorFlow recommends for implementing custom layers. The `__init__` method, constructor of the `EulerIntegratorCell`, assigns the constants C and m as well as the initial state a_0. This method also creates an attribute `dKlayer` to the object of `EulerIntegratorCell`. As we will detail later, this is an interesting feature that will essentially allow us to specify any model to `dKlayer`. Although `dKlayer` can be implemented using physics, as we discussed before, we will illustrate the case in which `dKlayer` is a multilayer perceptron. The `call` method effectively implements Eq. (5). With regards to numerically integrating fatigue crack growth, we still have to implement Eq. (4).

Here, we will use the TensorFlow native recurrent neural network class, `RNN`, to effectively march in time; and therefore, implement Eq. (4). Listing 3 details how we can use an object from the `EulerIntegratorCell` and couple it with `RNN` to create a model ready to be trained. The function `create_model` takes `C`, `m`, `a0`, and `dKLayer` so that an `EulerIntegratorCell` object can be instantiated. Additionally, it also takes `batch_input_shape` and `return_sequences`. The variable `batch_input_shape` is used within `EulerIntegratorCell` to reinforce the shape of the inputs. Although `batch_input_shape` is not directly specified in `EulerIntegratorCell`, it belongs to `**kwargs` and it will be consumed in the constructor of `Layer`.

Eq. (4) starts to be implemented when `PINN`, the object of class `RNN`, is instantiated. As a recurrent neural network, `PINN` has the ability to march through time and execute the `call` method of the `euler` object. Lines 10 to 14 of List. 3 are needed so that an optimizer and loss function are linked to the model that will be created. In this example, we use the mean square error (`'mse'`) as loss function and RMSprop as an optimizer.

With `EulerIntegratorCell` and `create_model` defined, we can proceed to training and predicting with the hybrid physics-informed neural network model. Listing 4 details how to build the main portion of the Python script. From line 2 to line 9, we are simply defining the material properties and loading the data. After that, we can create the `dKlayer` model. Within TensorFlow, `Sequential` is used to create models that will be stacks of several layers. `Dense` is used to define a layer of a neural network. Line 12 initializes `dKlayer` preparing it to receive the different layers in sequence. Line 13 adds the first layer with 5 neurons (and `tanh` as activation function). Line 14 adds the second layer with 1 neuron. Creating the hybrid physics-informed neural network model is as simple as calling `create_model`, as shown in line 19. As is, `model` is ready to be trained, which is done in line 23. For the sake of the example though, we can check the predictions at the training set before and after the training (lines 22 and 24, respectively). The fact that we have to slice the third dimension of the array with `[:,:]` is simply an artifact of TensorFlow. The way the code is implemented, predictions are done by marching through time while integrating fatigue crack growth starting from `a0`. However, since we have set `return_sequences=False` (default in `create_model`), the predictions are returned only for the very last cycle. Setting that flag to `True` would change the behavior of the `predict_on_batch`, which would return the entire time series.

Fig. 3 illustrates the results obtained when running the codes within folder `first_order_ode` available at Fricke et al. (2020). Fig. 3(a) shows the history of the loss function (mean square error) throughout the training. The loss converges rapidly within the first ten epochs and shows minor further convergence in the following ten epochs. We would like to point out that experienced TensorFlow users could further customize the implementation to stop the hyperparameter optimization as loss function converges. Fig. 3(b) shows the prediction against actual fatigue crack length at the last loading cycle for a test set (data points not used to train the physics-informed neural network). While results may vary from run-to-run, given that `RMSprop` implements a stochastic gradient descend algorithm, it is clear that the hybrid physics-informed neural network was able to learn the latent (hidden) stress intensity range model. Finally, we repeated the training of the proposed physics-informed neural network 100 times so that we can study the repeatability of results. Fig. 3(c) shows the histograms of the mean squared error at both training and test sets. Most of the time, the mean squared error is below $20 \times 10^{-6}(\text{m})^2$, while it was never above $100 \times 10^{-6}(\text{m})^2$. Considering that the observed crack lengths are within 5×10^{-3} (m) to 35×10^{-3} (m), these values of mean square error are sufficiently small.

```python
# basic packages
import pandas as pd
import numpy as np

# keras essentials
from tensorflow.keras.layers import RNN, Dense, Layer
from tensorflow.keras import Sequential
from tensorflow.keras.optimizers import RMSprop

# tensorflow operators
from tensorflow.python.framework import tensor_shape
from tensorflow import float32, concat, convert_to_tensor
```

Listing 1: Import section for the Euler integration example.

```python
class EulerIntegratorCell(Layer):
  def __init__(self, C, m, dKlayer, a0, units=1, **kwargs):
    super(EulerIntegratorCell, self).__init__(**kwargs)
    self.units = units
    self.C = C
    self.m = m
    self.a0 = a0
    self.dKlayer = dKlayer

    ...

  def call(self, inputs, states):

    ...

    x_d_tm1 = concat((inputs,a_tm1[0,:]),axis=1)
    dk_t    = self.dKlayer(x_d_tm1)
    da_t    = self.C * (dk_t ** self.m)
    a       = da_t + a_tm1[0, :]
    return a, [a]
```

Listing 2: Euler integrator cell.

```python
def create_model(C,m,dKlayer,a0,batch_input_shape,return_sequences):
    euler = EulerIntegratorCell(C=C,m=m,dKlayer=dKlayer,a0=a0,
            batch_input_shape=batch_input_shape,return_state=False)
    PINN = RNN(cell=euler,batch_input_shape=batch_input_shape,
            return_sequences=return_sequences,return_state=return_state)
    model = Sequential()
    model.add(PINN)
    model.compile(loss='mse',optimizer=RMSprop(1e-2))
  return model
```

Listing 3: Create model function for the Euler integration example.

3.3. System of second order ordinary differential equations

In this section, we will focus on our hybrid physics-informed neural network implementation of a system of second order ordinary differential equations. In the case study, we will highlight the useful aspect of system identification. This is when observed data is used to estimate parameters of the governing equations.

Consider the system of second order ordinary differential equation expressed in the form

$$\mathbf{P}(t)\frac{d^2\mathbf{y}}{dt^2} + \mathbf{Q}(t)\frac{d\mathbf{y}}{dt} + \mathbf{R}(t)\mathbf{y} = \mathbf{u}(t) \tag{7}$$

where $\mathbf{u}(t)$ are controllable inputs to the system, $\mathbf{y}$ are the outputs of interest, and t is time. The solution to Eq. (7) depends on initial conditions ($\mathbf{y}$ as well as $\frac{d\mathbf{y}}{dt}$ at $t = 0$), input data ($\mathbf{u}(t)$ known at different time steps).

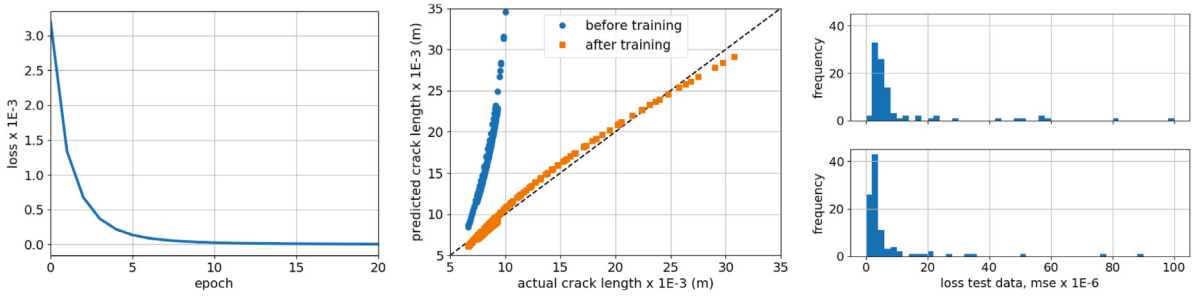
3.3.1. Case study: Forced vibration of 2-degree-of-freedom system

In this case study, we consider the motion for two masses linked together springs and dashpots, as depicted in Fig. 4(a). The number of degrees of freedom of a system is the number of independent coordinates necessary to define motion (equal to the number of masses in this case). Under such circumstances, the equations of are obtained using Newton's second law

$$\begin{aligned} &\mathbf{M}\ddot{\mathbf{y}} + \mathbf{C}\dot{\mathbf{y}} + \mathbf{K}\mathbf{y} = \mathbf{u}, \text{ or alternatively} \\ &\ddot{\mathbf{y}} = f(\mathbf{u}, \dot{\mathbf{y}}, \mathbf{y}) = \mathbf{M}^{-1}(\mathbf{u} - \mathbf{C}\dot{\mathbf{y}} - \mathbf{K}\mathbf{y}), \end{aligned} \tag{8}$$

```
if __name__ == "__main__":
    # Paris law coefficients
    [C, m] = [1.5E-11, 3.8]

    # data
    Stest  = np.asarray(pd.read_csv('./data/Stest.csv'))[:,:,np.newaxis]
    Strain = np.asarray(pd.read_csv('./data/Strain.csv'))[:,:,np.newaxis]
    atrain = np.asarray(pd.read_csv('./data/atrain.csv'))
    a0     = np.asarray(pd.read_csv('./data/a0.csv'))[0,0]*np.ones((Strain.shape[0],1))

    # stress-intensity layer
    dKlayer = Sequential()
    dKlayer.add(Dense(5, input_shape=(2,), activation='tanh'))
    dKlayer.add(Dense(1))

    ...

    # fitting physics-informed neural network
    model = create_model(C=C, m=m, dKlayer=dKlayer,
            a0=ops.convert_to_tensor(a0,dtype=float32),
            batch_input_shape=Strain.shape)
    aPred_before = model.predict_on_batch(Stest)[:,:]
    model.fit(Strain, atrain, epochs=20, steps_per_epoch=1, verbose=1)
    aPred = model.predict_on_batch(Stest)[:,:]
```

Listing 4: Training and predicting in the Euler integration example



(a) Loss function convergence

(b) Predicted vs. actual crack length at the test set

(c) Mean square error histogram of on train and validation data (training repeated 100 times).

Fig. 3. Euler integration results. After training is complete, the model-form uncertainty is greatly reduced. Trained model can be used directly for predictions outside the training set. We observe repeatability of results after repeating the training of the physics-informed neural network varying initialization of weights.

where:

$$\mathbf{M} = \begin{bmatrix} m_1 & 0 \\ 0 & m_2 \end{bmatrix}, \ \mathbf{C} = \begin{bmatrix} c_1 + c_2 & -c_2 \\ -c_2 & c_2 + c_3 \end{bmatrix}, \ \mathbf{K} = \begin{bmatrix} k_1 + k_2 & -k_2 \\ -k_2 & k_2 + k_3 \end{bmatrix},$$
$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix}, \text{ and } \mathbf{u} = \begin{bmatrix} u_1 \\ u_2 \end{bmatrix}. \tag{9}$$

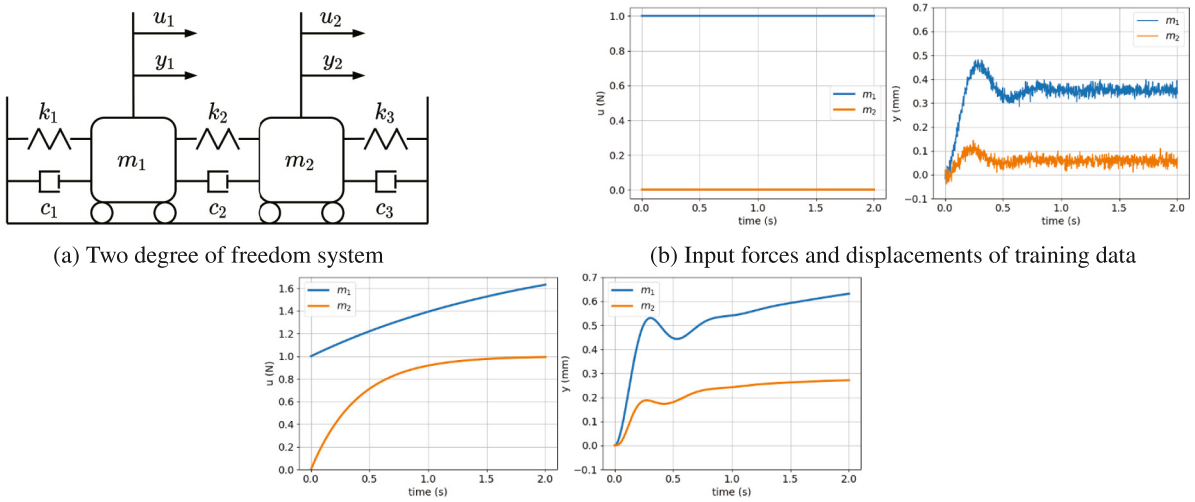
We assume that while the masses and spring coefficients are known, the damping coefficients are not. Once these coefficients are estimated based on available data, the equations of motion can be used for predicting the mass displacements given any input conditions (useful for design of vibration control strategies, for example).

Fig. 4(b) and 4(c) illustrate the data used in this case study. Here, we used $m_1 = 20$ (kg), $m_2 = 10$ (kg), $c_1 = 30$ (N.s/m), $c_2 = 5$ (N.s/m), $c_3 = 10$ (N.s/m), $k_1 = 2\times10^3$ (N/m), $k_2 = 1\times10^3$ (N/m), and $k_3 = 5\times10^3$ (N/m) to generate the data. On the training data, a constant force $u_1(t) = 1$ (N) is applied to mass m_1, while m_2 is let free. On the test data, time-varying forces are applied to both masses. The displacements of both masses are observed every 0.002 (s) for two seconds. The observed displacements of the training data are contaminated with Gaussian noise with zero mean and 1.5×10^{-5} standard deviation.

3.3.2. Computational implementation

Within folder `second_order_ode` of the repository available at Fricke et al. (2020), the interested reader will find the training and test data in the `data.csv` and `data02.csv` files, respectively. The time stamp is given by column `t`. The input forces are given by columns `u1` and `u2`. The measured displacements are given by columns `yT1` and `yT2`. Finally, the actual (but unknown) displacements are given by columns `y1` and `y2`.

With defined initial conditions $\mathbf{y}(t = 0) = \mathbf{y}_0$ and $\dot{\mathbf{y}}(t = 0) = \dot{\mathbf{y}}_0$, we can use the classic Runge–Kutta method (Press et al., 2007; Butcher and Wanner, 1996) to numerically integrate Eq. (8) over time set with time



(a) Two degree of freedom system

(b) Input forces and displacements of training data

(c) Input forces and displacements of test data

Fig. 4. Forced vibration details. Response of a two degree of freedom system is a function of input forces applied at the two masses. Training data is contaminated with Gaussian noise (emulating noise in sensor reading). Test data is significantly different from training data.

step h:

$$
\begin{aligned}
&\begin{bmatrix}\dot{\mathbf{y}}_{n+1}\\ \mathbf{y}_{n+1}\end{bmatrix} = \begin{bmatrix}\dot{\mathbf{y}}_{n}\\ \mathbf{y}_{n}\end{bmatrix} + h\sum_i b_i \boldsymbol{\kappa}_i\,, \qquad \boldsymbol{\kappa}_i = \begin{bmatrix}\mathbf{k}_i\\ \bar{\mathbf{k}}_i\end{bmatrix},\\
&\mathbf{k}_1 = f(\mathbf{u}_n, \dot{\mathbf{y}}_n, \mathbf{y}_n),\ \bar{\mathbf{k}}_1 = \mathbf{y}_n\\
&\mathbf{k}_i = f\left(\mathbf{u}_{n+c_i h}, \dot{\mathbf{y}}_n + h\sum_j^{i-1} a_{ij}\mathbf{k}_j, \mathbf{y}_n + h\sum_j^{i-1} a_{ij}\bar{\mathbf{k}}_j\right),\\
&\bar{\mathbf{k}}_i = \mathbf{y}_n + h\sum_j^{i-1} a_{ij}\bar{\mathbf{k}}_j,\\
&\mathbf{A} = \begin{bmatrix}0&0&0&0\\ 1/2&0&0&0\\ 0&1/2&0&0\\ 0&0&1&0\end{bmatrix}, \qquad \mathbf{b} = \begin{bmatrix}1/6\\ 1/3\\ 1/3\\ 1/6\end{bmatrix}, \qquad \mathbf{c} = \begin{bmatrix}0\\ 1/2\\ 1/2\\ 1\end{bmatrix},
\end{aligned}
\tag{10}
$$

In this section, we will show how we can use observed data to tune specific coefficients in Eq. (8). Specifically, we will tune the damping coefficients c_1, c_2, and c_3 by minimizing the mean squared error:

$$
\Lambda = \frac{1}{n}(\mathbf{y} - \hat{\mathbf{y}})^T(\mathbf{y} - \hat{\mathbf{y}}), \tag{11}
$$

where n is the number of observations, $\mathbf{y}$ are observed displacements, and $\hat{\mathbf{y}}$ are the displacements predicted using the physics-informed neural network.

We will use all the packages shown Listing 1, in addition to `linalg` imported from `tensorflow` (we did not show a separate listing to avoid clutter). Listing 5 shows the important snippets of implementation of the Runge–Kutta integrator cell (to avoid clutter, we leave out the lines that are needed for data-type reinforcement). The `__init__` method, constructor of the `RungeKuttaIntegratorCell`, assigns the mass, stiffness, and damping coefficient initial guesses, as well as the initial state and Runge–Kutta coefficients. The `call` method effectively implements Eq. 4 while the `_fun` method implements Eq. (8).

Listing 6 details how we use objects from `RungeKuttaIntegratorCell` and `RNN` to create a model ready to be trained. The function `create_model` takes `m`, `c`, and `k` arrays, `dt`, `initial_state`, `batch_input_shape` and `return_sequences` so that a `RungeKuttaIntegratorCell` object is instantiated. Parameter `batch_input_shape` is used within `RungeKuttaIntegratorCell` to reinforce the shape of the inputs (although it is not directly specified in `RungeKuttaIntegratorCell`, it belongs to `**kwargs` and it will be consumed in the constructor of `Layer`).

Similarly to the Euler example, we will use the TensorFlow native recurrent neural network class, `RNN`, to effectively march in time. The march through time portion of Eq. (8) starts to be implemented when `PINN`, the object of class `RNN`, is instantiated. As a recurrent neural network, `PINN` has the ability to march through time and execute the `call` method of the `rkCell` object. Lines 9 to 11 of List. 6 are needed so that an optimizer and loss function are linked to the model that will be created. In this example, we use the mean square error (`'mse'`) as loss function and RMSprop as an optimizer.

Listing 7 details how to build the main portion of the Python script. From line 2 to line 5, we are simply defining the masses, spring coefficients (which are assumed to be known), as well as damping coefficients, which are unknown and will be fitted using observed data (here, values only represent an initial guess for the hyperparameter optimization). Creating the hybrid physics-informed neural network model is as simple as calling `create_model`, as shown in line 16. As is, `model` is ready to be trained, which is done in line 19. or the sake of the example though, we can check the predictions at the training set before and after the training (lines 18 and 20, respectively).

Fig. 5 illustrates the results obtained when running the codes within folder `second_order_ode` available at Fricke et al. (2020). Fig. 5(a) shows the history of the loss function (mean square error) throughout the training. Figs. 5(b) and 5(c) show the prediction against actual displacements. Similarly to the Euler case study, results may vary from run-to-run, depending on the initial guess for c_1, c_2, and c_3 as well as performance of `RMSprop`. The loss converges rapidly within 20 epochs and only marginally further improves after 40 epochs. As illustrated in Fig. 5(b), the predictions converge to the observations, filtering the noise in the data. Fig. 5(c) shows that the model parameters identified after training the model allowed for accurate predictions on the test set. In order to further evaluate the performance of the model, we created contaminated training data sets where we emulate the case that sensors used to read the output displacement exhibit a burst of high noise levels at different points in the time series. For example, Fig. 5(d) illustrates the case in which the burst of high noise level happens between 0.5 (s) and 0.75 (s); while in Fig. 5(e), this data corruption happened at two different time periods (0.1 to 0.2 (s) and 0.4 to 0.5 (s)). In both cases, model parameters identified after training the model allowed for accurate predictions. Noise in the data imposes a challenge for model parameter identification. Table 1 lists the identified parameters for the separate model training runs with and without the bursts of corrupted data. As expected, c_1 is easier to identify, since it is connected between the wall and m_1, which is twice as large as m_2. On top of that, the

```
class RungeKuttaIntegratorCell(Layer):
    def __init__(self, m, c, k, dt, initial_state, **kwargs):
        super(RungeKuttaIntegratorCell, self).__init__(**kwargs)
        self.Minv = linalg.inv(np.diag(m))
        self._c   = c
        self.K    = self._getCKmatrix(k)
        self.A  = np.array([0., 0.5, 0.5, 1.0], dtype='float32')
        self.B  = np.array([[1/6, 2/6, 2/6, 1/6]], dtype='float32')
        self.dt = dt
        ...

    def build(self, input_shape, **kwargs):
        self.kernel = self.add_weight("C",shape=self._c.shape,
            trainable=True, initializer=lambda shape,dtype: self._c,
            **kwargs)
        self.built  = True

    def call(self, inputs, states):
        C    = self._getCKmatrix(self.kernel)
        y    = states[0][:,:2]
        ydot = states[0][:,2:]

        yddoti = self._fun(self.Minv, self.K, C, inputs, y, ydot)
        yi     = y + self.A[0] * ydot * self.dt
        ydoti  = ydot + self.A[0] * yddoti * self.dt
        fn     = self._fun(self.Minv, self.K, C, inputs, yi, ydoti)
        for j in range(1,4):
            yn    = y + self.A[j] * ydot * self.dt
            ydotn = ydot + self.A[j] * yddoti * self.dt
            ydoti = concat([ydoti, ydotn], axis=0)
            fn    = concat([fn,self._fun(self.Minv,self.K,C,inputs,yn,ydotn)],axis=0)

        y    = y + linalg.matmul(self.B, ydoti) * self.dt
        ydot = ydot + linalg.matmul(self.B, fn) * self.dt
        return y, [concat(([y, ydot]), axis=-1)]

    def _fun(self, Minv, K, C, u, y, ydot):
        return linalg.matmul(u - linalg.matmul(ydot, C, transpose_b=True) - linalg.matmul
    (y, K, transpose_b=True), Minv, transpose_b=True)
        ...
```

Listing 5: Runge–Kutta integrator cell.

```
def create_model(m,c,k,dt,initial_state,batch_input_shape,
                 return_sequences=True, unroll=False):
    rkCell = RungeKuttaIntegratorCell(m=m,c=c,k=k,dt=dt,
            initial_state=initial_state)
    PINN = RNN(cell=rkCell,batch_input_shape=batch_input_shape,
            return_sequences=return_sequences,
            return_state=False,unroll=unroll)
    model = Sequential()
    model.add(PINN)
    model.compile(loss='mse',optimizer=RMSprop(1e4),metrics=['mae'])
    return model
```

Listing 6: Create model function for the Runge–Kutta integration example.

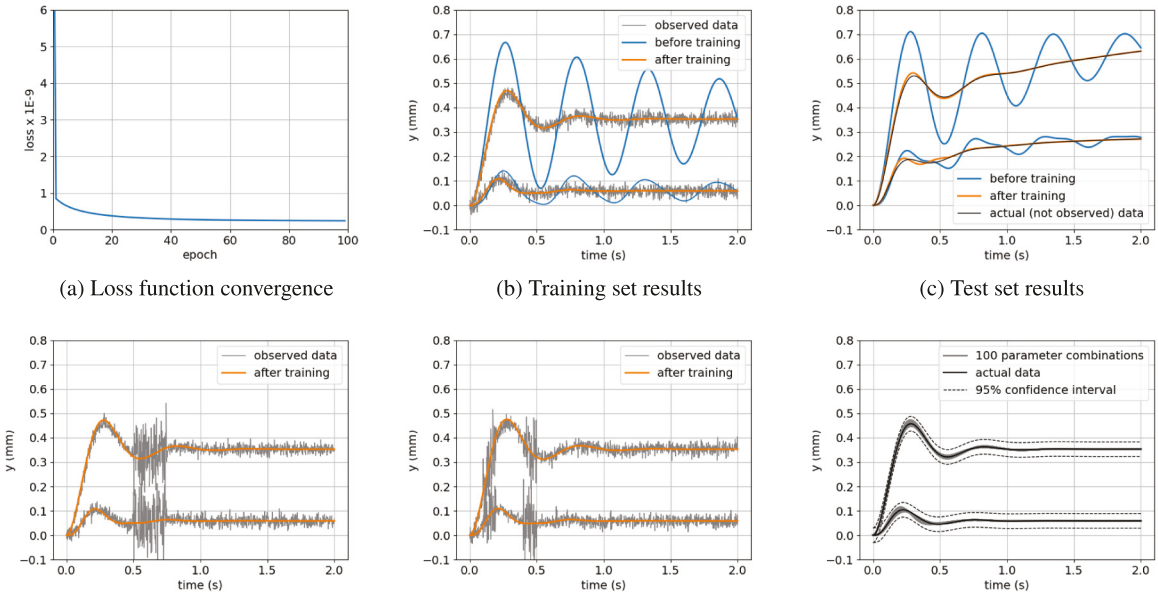
force is applied in m_1. In this particular example, the outputs show low sensitivity to c_2 and c_3. Fig. 5(f) show a comparison between the actual training data (in the form of mean and 95% confidence interval) and the predicted curves when $70 \leq c_2 \leq 110$ and $15 \leq c_3 \leq 120$. Despite the apparently large ranges for c_2 and c_3, their influence in the variation of the predicted output is still smaller than the noise in the data.

4. Summary and closing remarks

In this paper, we discussed Python implementations of ordinary differential equation solvers using recurrent neural networks with customized repeatable cells with hybrid implementation of physics-informed kernels. In our examples, we found that this approach is useful for both quantification of model-form uncertainty as well as

```python
if __name__ == "__main__":
    # masses, spring coefficients, and damping coefficients
    m = np.array([20.0, 10.0],dtype='float32')
    c = np.array([10.0, 10.0, 10.0],dtype='float32') # initial guess
    k = np.array([2e3, 1e3, 5e3],dtype='float32')

    # data
    df = pd.read_csv('./data/data.csv')
    t  = df[['t']].values
    dt = (t[1] - t[0])[0]
    utrain = df[['u0','u1']].values[np.newaxis,:,:]
    ytrain = df[['yT0','yT1']].values[np.newaxis,:,:]

    # fitting physics-informed neural network
    initial_state = np.zeros((1,2*len(m),),dtype='float32')
    model = create_model(m,c,k,dt,initial_state=initial_state,
        batch_input_shape=utrain.shape)
    yPred_before = model.predict_on_batch(utrain)[0,:,:]
    model.fit(utrain, ytrain, epochs=100, steps_per_epoch=1, verbose=1)
    yPred = model.predict_on_batch(utrain)[0,:,:]
```

Listing 7: Training and predicting in the Runge–Kutta integration example



(a) Loss function convergence

(b) Training set results

(c) Test set results

(d) Prediction at first contaminated training set

(e) Prediction at second contaminated training set

(f) Damping coefficient variation. Confidence interval based on Gaussian noise.

Fig. 5. Runge–Kutta integration results. After training, damping coefficients are identified (Table 1). Model can be used in test cases that are completely different from training. Due to nature of physics that governs the problem, responses are less sensitive to coefficients c_2 and c_3, when compared to c_1. Nevertheless, model identification is successful even when noise level varies throughout training data.

Table 1
Identified damping coefficients. Actual values for the coefficients are $c_1 = 100.0$, $c_2 = 110.0$, and $c_3 = 120.0$. Due to nature of physics that governs the problem, responses are less sensitive to coefficients c_2 and c_3, when compared to c_1 (Fig. 5(f)).

Noise in observed output	c_1	c_2	c_3
Gaussian	115.1	71.6	16.7
Gaussian with single burst of high contamination	113.2	70.0	17.1
Gaussian with double burst of high contamination	109.2	70.7	15.3

model parameter estimation. We demonstrated our framework on two examples:

- *Euler integration of fatigue crack propagation:* our hybrid model framework characterized model form uncertainty regarding the stress intensity range used in Paris law. We implemented the numerical integration of the first order differential equation through the Euler method given that the time history of far-field stresses is available. For this case study, we observed good repeatability of results with regards to variations in initialization of the neural network hyperparameters.
- *Runge–Kutta integration of a 2 degree-of-freedom vibrations system:* our hybrid approach is capable of model parameter identification. We implemented the numerical integration of the second order differential equation through the Runge–Kutta method

given that the physics is known and both inputs and outputs are observed through time. For this case study, we saw that the identified model parameters led to accurate prediction of the system displacements.

In this paper, we demonstrated the ability to directly implement physics-based models into a hybrid neural network and leverage the graph-based modeling capabilities found in platforms such as Tensorflow. Specifically, our implementation inherits the capabilities offered by these frameworks such as implementation of recurrent neural network base class, automatic differentiation, and optimization methods for hyperparameter optimization. The examples presented here as well as source codes are all open-source under the MIT License and are available in the GitHub repository https://github.com/PML-UCF/pinn_code_tutorial.

In terms of future work, we believe there are many opportunities to advance the implementation as well as the application of the demonstrated approach. For example, one can explore further aspects of parallelization, potentially extending implementation to low-level CUDA implementation (TensorFlow Contributors, 2020). Another interesting aspect to explore would be data batching and dropout (Srivastava et al., 2014), which could be particularly useful when dealing with large datasets. In terms of applications, we believe that hybrid implementations like the ones we discussed are beneficial when reduced order models can capture part of the physics. Then data-driven models can compensate for the remaining uncertainty and reduce the gap between predictions and observations. In the immediate future, it would be interesting to see applications in dynamical systems and controls. For example, Altan and collaborators proposed a new model predictive controller for target tracking of a three-axis gimbal system (Altan and Hacioglu, 2020) as well as a real-time control system for UAV path tracking (Altan et al., 2018). The UAV control system is based on a nonlinear auto-regressive exogenous neural network where the proposed model predictive controller for the gimbal system is based on a Hammerstein model. The proposed control methods are used for real-time target tracking of a moving target under influence from external disturbances. It would be interesting to see how our proposed approach can be incorporated in real-time controls.

CRediT authorship contribution statement

Renato G. Nascimento: Methodology, Software, Formal analysis, Investigation, Data curation, Writing, Visualization. **Kajetan Fricke:** Methodology, Software, Formal analysis, Investigation, Data curation, Writing, Visualization. **Felipe A.C. Viana:** Conceptualization, Methodology, Validation, Software, Formal analysis, Investigation, Writing, Supervision, Funding acquisition.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Appendix. Multilayer perceptrons and recurrent neural networks

Fig. 6 illustrates the popular multilayer perceptron. Each layer can have one or more perceptrons (nodes in the graph). A perceptron applies a linear combination to the input variables followed by an activation function

$$v = f(z) \quad \text{and} \quad z = \mathbf{w}^T \mathbf{u} + b, \tag{12}$$

where v is the perceptron output; $\mathbf{u}$ are the inputs; $\mathbf{w}$ and b are the perceptron hyperparameters; and $f(.)$ is the activation function. Throughout this paper, we used the hyperbolic tangent (tanh), sigmoid and the exponential linear unit (elu) activation functions (although

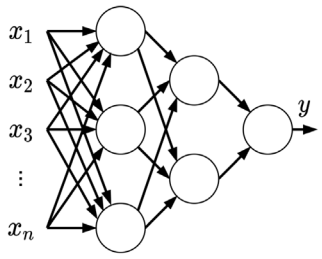


Fig. 6. Multilayer perceptron.

others could also be used, such as the rectified exponential linear unit):

$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}, \quad \text{sigmoid}(z) = \frac{1}{1 + e^{-z}},$$
$$\text{elu}(z) = \begin{cases} z & when z > 0, \text{ and} \\ e^z - 1 & otherwise. \end{cases} \tag{13}$$

The choice of number of layers, number of neurons in each layer, and activation functions is outside the scope of this paper. Depending on computational cost associated with application, we even encourage the interested reader to pursue neural architecture search (Kandasamy et al., 2018; Liu et al., 2018; Elsken et al., 2019) for optimization of the data-driven portions of the model.

References

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D.G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., Zheng, X., 2016. Tensorflow: A system for large-scale machine learning. In: 12th USENIX Symposium on Operating Systems Design and Implementation, OSDI. pp. 265–283.

Altan, A., Aslan, O., Hacioglu, R., 2018. Real-time control based NARX neural networks of hexarotor UAV with load transporting system for path tracking. In: 2018 6th International Conference on Control Engineering Information Technology. CEIT, IEEE, Istanbul, Turkey, pp. 1–6. http://dx.doi.org/10.1109/CEIT.2018.8751829.

Altan, A., Hacioglu, R., 2020. Model predictive control of three-axis gimbal system mounted on UAV for real-time target tracking under external disturbances. Mech. Syst. Signal Process. 138, http://dx.doi.org/10.1016/j.ymssp.2019.106548.

Baydin, A.G., Pearlmutter, B.A., Radul, A.A., Siskind, J.M., 2018. Automatic differentiation in machine learning: a survey. J. Mach. Learn. Res. 18 (153), 1–43.

Butcher, J., Wanner, G., 1996. Runge–Kutta methods: some historical notes. Appl. Numer. Math. 22 (1), 113–151. http://dx.doi.org/10.1016/S0168-9274(96)00048-7, Special Issue Celebrating the Centenary of Runge-Kutta Methods.

Chen, T.Q., Rubanova, Y., Bettencourt, J., Duvenaud, D.K., 2018. Neural ordinary differential equations. In: Bengio, S., Wallach, H., Larochelle, H., Grauman, K., Cesa-Bianchi, N., Garnett, R. (Eds.), 31st Advances in Neural Information Processing Systems. Curran Associates, Inc., pp. 6572–6583.

Cheng, Y., Huang, Y., Pang, B., Zhang, W., 2018. ThermalNet: A deep reinforcement learning-based combustion optimization system for coal-fired boiler. Eng. Appl. Artif. Intell. 74, 303–311. http://dx.doi.org/10.1016/j.engappai.2018.07.003.

Cho, K., Van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., Bengio, Y., 2014. Learning phrase representations using RNN encoder-decoder for statistical machine translation. arXiv preprint arXiv:1406.1078.

Chollet, F., et al., 2015. Keras. https://keras.io.

Collins, J.A., 1993. Failure of Materials in Mechanical Design: Analysis, Prediction, Prevention. John Wiley & Sons.

Connor, J.T., Martin, R.D., Atlas, L.E., 1994. Recurrent neural networks and robust time series prediction. IEEE Trans. Neural Netw. 5 (2), 240–254. http://dx.doi.org/10.1109/72.279188.

Dourado, A., Viana, F.A.C., 2020. Physics-informed neural networks for missing physics estimation in cumulative damage models: a case study in corrosion fatigue. ASME J. Comput. Inf. Sci. Eng. 20 (6), 061007. http://dx.doi.org/10.1115/1.4047173, 10.

Dowling, N.E., 2012. Mechanical Behavior of Materials: Engineering Methods for Deformation, Fracture, and Fatigue. Pearson.

Elsken, T., Metzen, J.H., Hutter, F., 2019. Neural architecture search: a survey. J. Mach. Learn. Res. 20 (55), 1–21.

Fricke, K., Nascimento, R.G., Viana, F.A.C., 2020. Python Implementation of Ordinary Differential Equations Solvers using Hybrid Physics-informed Neural Networks. Zenodo, http://dx.doi.org/10.5281/zenodo.3895408, URL: https://github.com/PML-UCF/pinn_ode_tutorial.

Goodfellow, I., Bengio, Y., Courville, A., 2016. Deep Learning. MIT Press, URL: http://www.deeplearningbook.org.

Graves, A., Mohamed, A., Hinton, G., 2013. Speech recognition with deep recurrent neural networks. In: IEEE International Conference on Acoustics, Speech and Signal Processing. pp. 6645–6649. http://dx.doi.org/10.1109/ICASSP.2013.6638947.

Hochreiter, S., Schmidhuber, J., 1997. Long short-term memory. Neural Comput. 9 (8), 1735–1780. http://dx.doi.org/10.1162/neco.1997.9.8.1735.

Kandasamy, K., Neiswanger, W., Schneider, J., Poczos, B., Xing, E.P., 2018. Neural architecture search with Bayesian optimisation and optimal transport. In: Bengio, S., Wallach, H., Larochelle, H., Grauman, K., Cesa-Bianchi, N., Garnett, R. (Eds.), Advances in Neural Information Processing Systems, Vol. 31. Curran Associates, Inc., pp. 2016–2025.

Karpatne, A., Atluri, G., Faghmous, J.H., Steinbach, M., Banerjee, A., Ganguly, A., Shekhar, S., Samatova, N., Kumar, V., 2017. Theory-guided data science: A new paradigm for scientific discovery from data. IEEE Trans. Knowl. Data Eng. 29 (10), 2318–2331. http://dx.doi.org/10.1109/TKDE.2017.2720168.

Liu, C., Zoph, B., Neumann, M., Shlens, J., Hua, W., Li, L.-J., Fei-Fei, L., Yuille, A., Huang, J., Murphy, K., 2018. Progressive neural architecture search. In: The European Conference on Computer Vision, ECCV.

Nascimento, R.G., Viana, F.A.C., 2020. Cumulative damage modeling with recurrent neural networks. AIAA J. http://dx.doi.org/10.2514/1.J059250, Online First.

Pan, S., Duraisamy, K., 2020. Physics-informed probabilistic learning of linear embeddings of nonlinear dynamics with guaranteed stability. SIAM J. Appl. Dyn. Syst. 19 (1), 480–509. http://dx.doi.org/10.1137/19M1267246.

Pang, G., Karniadakis, G.E., 2020. Physics-informed learning machines for partial differential equations: Gaussian processes versus neural networks. In: Nonlinear Systems and Complexity. Springer International Publishing, pp. 323–343. http://dx.doi.org/10.1007/978-3-030-44992-6_14.

Paris, P., Erdogan, F., 1963. A critical analysis of crack propagation laws. J. Basic Eng. 85 (4), 528–533. http://dx.doi.org/10.1115/1.3656900.

Press, W.H., Teukolsky, S.A., Vetterling, W.T., Flannery, B.P., 2007. Numerical Recipes: The Art of Scientific Computing. Cambridge University Press, New York, USA.

Raissi, M., Karniadakis, G.E., 2018. Hidden physics models: machine learning of nonlinear partial differential equations. J. Comput. Phys. 357, 125–141. http://dx.doi.org/10.1016/j.jcp.2017.11.039.

Raissi, M., Perdikaris, P., Karniadakis, G., 2019. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. J. Comput. Phys. 378, 686–707. http://dx.doi.org/10.1016/j.jcp.2018.10.045.

Sak, H., Senior, A., Beaufays, F., 2014. Long short-term memory recurrent neural network architectures for large scale acoustic modeling. In: Fifteenth Annual Conference of the International Speech Communication Association. Singapore. pp. 338–342. https://www.isca-speech.org/archive/interspeech_2014/i14_0338.html.

Shen, C., Qi, Y., Wang, J., Cai, G., Zhu, Z., 2018. An automatic and robust features learning method for rotating machinery fault diagnosis based on contractive autoencoder. Eng. Appl. Artif. Intell. 76, 170–184. http://dx.doi.org/10.1016/j.engappai.2018.09.010.

Singh, S.K., Yang, R., Behjat, A., Rai, R., Chowdhury, S., Matei, I., 2019. PI-LSTM: Physics-infused long short-term memory network. In: 2019 18th IEEE International Conference on Machine Learning and Applications. ICMLA, IEEE, Boca Raton, USA, pp. 34–41. http://dx.doi.org/10.1109/ICMLA.2019.00015.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., Salakhutdinov, R., 2014. Dropout: A simple way to prevent neural networks from overfitting. J. Mach. Learn. Res. 15 (56), 1929–1958.

Sutskever, I., Martens, J., Hinton, G., 2011. Generating text with recurrent neural networks. In: Getoor, L., Scheffer, T. (Eds.), 28th International Conference on Machine Learning. ACM, Bellevue, USA, pp. 1017–1024, URL: https://icml.cc/2011/papers/524_icmlpaper.pdf.

TensorFlow Contributors, 2020. Create an op. URL: https://www.tensorflow.org/guide/create_op.

Viana, F.A.C., Nascimento, R.G., Yucesan, Y., Dourado, A., 2019. Physics-Informed Neural Networks Package. Zenodo, http://dx.doi.org/10.5281/zenodo.3356877, URL: https://github.com/PML-UCF/pinn.

Yucesan, Y.A., Viana, F.A.C., 2020. A physics-informed neural network for wind turbine main bearing fatigue. Int. J. Progn. Health Manag. 11 (1), 27–44.